信息流数据仓库

(作者：罗圣杰)

# 第1章 项目框架

## 1.1 概述

随着人们对手机依赖程度的增加，手机已经成为了绝大多数人社交、购物、休闲娱乐、学习、发表见解、获取时事新闻等需求的主要实现渠道。手机是绝大数用户流量的入口

* 当一款APP上线后，为了更好地改进这款APP，公司需要了解到这款APP的详细使用情况，例如新增用户数、活跃用户数等；为了了解用户对于这款APP不同版本的接受程度，需要了解这款APP在用户群体中的版本分布情况。
* 目前移动端作为手机的巨大入口，一些工具类的app用户停留时长不高。为了延长用户在手机上的停留时长，方法之一是在app中嵌入内容。信息流包括音频，视频，文字，图片等。丰富的内容，能极大限度的留住用户。这需要对用户进行个性化分析，所以信息流推荐系统就起到非常大的作用。

以上这些需求的实现，都要建立在一个稳定的分布式日志采集和统计分析系统之上。

在大数据技术趋于成熟的今天，大数据技术的发展使采集海量用户信息并分析用户行为进而有目的的改进APP这一需求的实现成为可能，本项目就致力于打造一套成熟的日志数据统计分析系统，通过离线数据分析系统和实时数据分析系统两个模块的协同作用，完成APP各项指标的分析，同时为信息流提供相关的数据支持与实现。

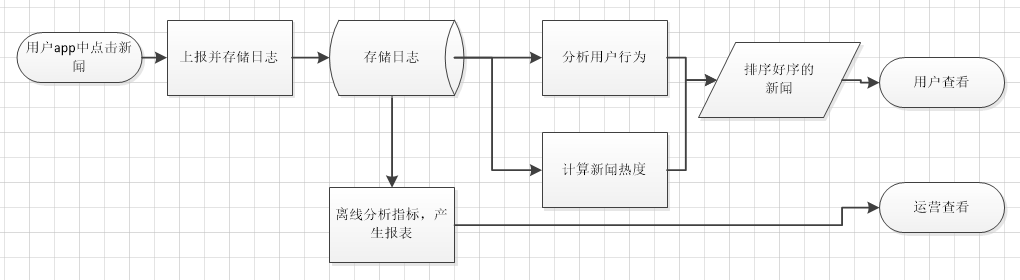
## 1.2 项目需求

### 1.2.1 日志业务流程

现在我们需要这样一个系统，来承载信息流，实现下面的功能：

分析用户行为功能：

1. 收集用户观看新闻的日志
2. 每日分析用户的行为
3. 实时对用户的行为作出反馈和推荐



图为业务流程

**对象说明：**

* 用户:用户由设备上传的唯一ID辨别。Id由前端产生，用来标注每一个用户。
* 新闻:包括图片，文字组合成的新闻，带有唯一id，计算过程以id处理
* 新闻质量:与点击率相关联
* 用户中心：数据仓库中维护所有点击过新闻的用户。
* 新闻中心：数据仓库中维护所有新闻的集合



图为数据流程

**报表说明：**

* 新增用户：首次联网使用应用的用户。如果一个用户首次打开某app，那这个用户定义为新增用户；卸载再安装的设备，不会被算作一次新增。新增用户包括日新增用户、周新增用户、月新增用户。
* 活跃用户：打开应用的用户即为活跃用户，不考虑用户的使用情况。每天一台设备打开多次会被计为一个活跃用户。
* 月活跃用户：某个自然周（月）内启动过应用的用户，该周（月）内的多次启动只记一个活跃用户。

### 1.2.2 离线报表需求

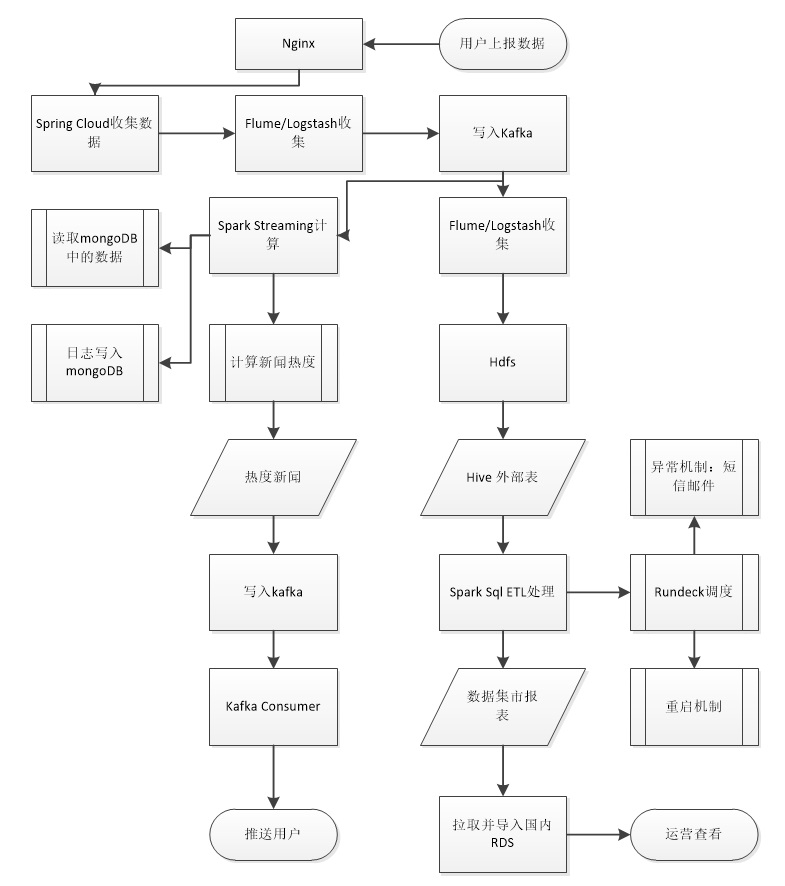
* 月活跃用户，日活跃用户
* 新增用户
* 新闻每日点击率，展示量，点击量
* 活跃用户信息（用户最近3天，7天，30天对新闻产生点击）
* 历史用户信息
* 历史新闻信息（最前和最后展示时间）

### 1.2.3 实时报表需求

* 最近24小时新闻点击率
* 根据点击计算热度，根据个性化排序，推送给后台。

## 1.3 项目框架

### 1.3.1 数据仓库流程



图：整个数据仓库的数据流程

**注意：**

1. **其中MongoDb可以使用Hbase**
2. **Logstash与Flume可以互换**
3. **Spring Cloud是Tomcat的升级版本，也可以用tomcat+Javaweb**
4. **Flume，kafka，spark可以使用老版本**

根据图中的项目框架可知，本项目由离线数据处理系统和实时数据处理系统两个部分组成。

离线数据处理系统与实时数据处理系统共用一套日志采集系统，日志采集系统采用了双层Flume拓扑结构，第一层实现数据的采集，第二层实现数据的集中聚合处理；手机APP客户端的日志数据被日志采集系统采集完成后，被分别输送到离线和实时数据处理系统中进行处理。

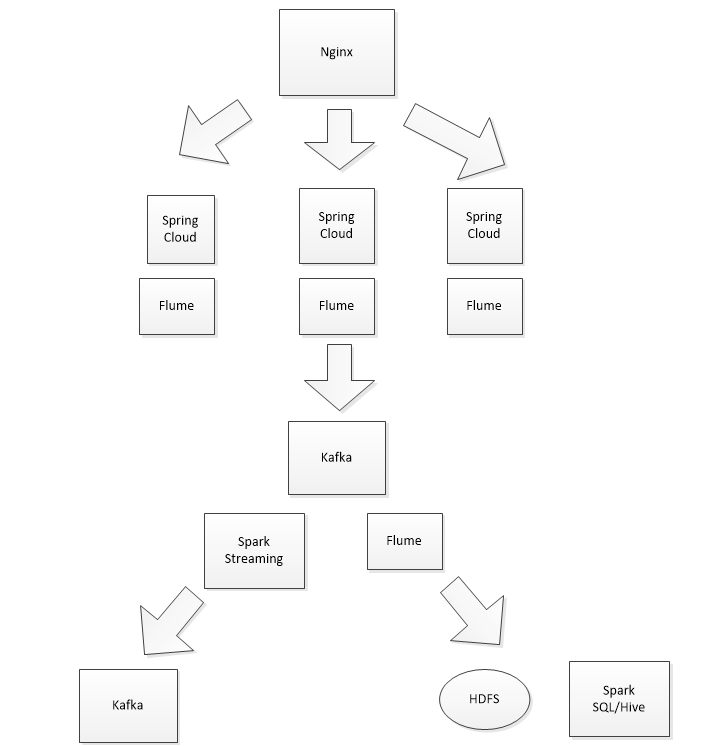
首先，APP客户端的日志数据定时（如0.5小时/次）服务器发送日志，服务器端通过Nginx实现负载均衡，Nginx将日志数据负载均衡到多给web应用上，然后通过日志写在磁盘。

随后，双层Flume架构中的第一层数据采集Flume将对应的Tomcat生成的日志文件采集到其拓扑结构中，随后多个第一层数据采集Flume的数据汇总到第二层的数据聚合Flume上，完成数据的聚合和集中处理。

然后，第二层的数据聚合Flume根据日志数据的类型，将日志数据发送到不同的Kafka主题中，在Kafka中完成数据的分布式存储。

最后，离线数据处理系统和实时数据处理系统分别从Kafka中消费消息，各自完成数据的离线分析处理和实时分析处理。

### 1.3.2 技术组件



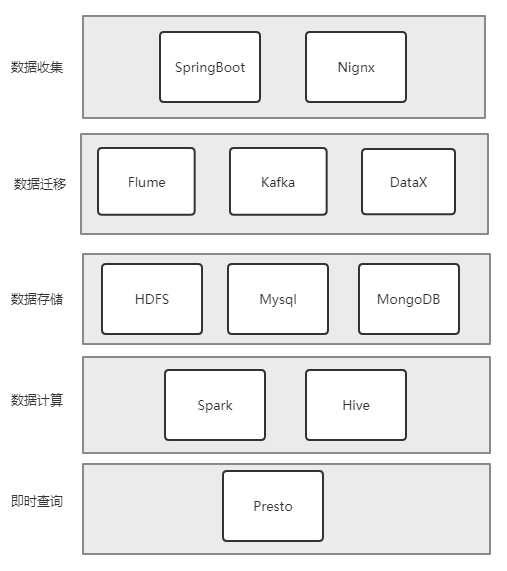
数据收集层组件

用户在使用APP的过程中，会不断产生日志数据。当用户后台运行的APP的时候，会产生后台日志。打开APP的时候，会产生前台APP启动日志，启动日志中记录了APP启动时间，运行时长等信息；

APP中运行的服务会定时将手机系统中产生的日志提取出来，然后经过一定的处理，例如合并等，目的是减少对服务端的压力，数据合并完成后，**在APP中有一个数据库，合并的数据会放入这一数据库中**，因为用户有可能断网，导致数据不能及时的发送到服务器，因此先存储在APP的本地数据库中，等到用户联网的时候再一并发送出来。

当手机客户端需要发送数据时，会对数据进行对称加密和压缩，压缩可以减少服务器的带宽，然后调用服务器的接口上传数据，服务端对数据进行解压和解密，然后用SLF4J写入本地磁盘中。（**写入本地磁盘的目的是让业务系统与采集系统完全解耦**）

**所以有几类产品：**



* 数据集成:Flume，DataX，Kafka
* 数据计算:Spark，Hive
* 数据存储:HDFS，S3，Mysql，Mongo
* 数据管理:Rundeck，元数据管理
* 数据查询:Presto

**数仓设计的几个考虑点：**

* 容灾性：数据备份
* 高可用: 分布式应用。一些应用宕机之后能很快恢复
* 即时性：流式计算的处理时间需要在业务要求的时间以内。
* 预警：有相关的检查，短信，电话预警
* 文档：包括元数据，注释或电子文档，对数据有相关说明

### 1.3.3 数据集市

数据集市就是数据仓库的产品。运营有时候需要进行一些自定义的查询，或报表查看这个时候需要看一些可视化的报表。这个时候就要获取数据集市的数据产品。可以分为:

* 可视化与非可视化数据报表
* 即时查询与周期性查询数据报表

## 1.4 原始日志概述

本项目中的日志数据由三种类型的日志组成，分别为

* 新闻点击/展示日志。
* 用户后台启动日志
* 用户前台启动日志

三种日志中共有的数据内容是：userId、area、appVersion,appKey。

### 1.4.1 新闻点击/展示日志

格式如下：

1537619705000| **{**

**"appkey"**: "browser"**,**

**"he"**: **{**

**"userId"**: "a12234sd"**,**

**"area"**: "BR"**,**

**"appVersion"**: "V1.0.2",

**"appTime"**: "1537533305000"

**},**

**"et"**: [**{**

**"eventName "**: "display"**,**

**"kv "**: **{**

**"action"**: "2"，**"newsId"**: "2343234"

**}**

**}]**

**}**

|  |  |
| --- | --- |
| 标签 | 含义 |
| 最前面的时间 | 服务端加上的时间 |
| appkey | APP的ID |
| he | 公共字段集合 |
| userId | 用户唯一Id，根据imei生成 |
| area | 国家 |
| appVerson | app版本号 |
| appTime | 客户端日志产生时间 |
| et | 业务字段集合，是一个json数组，多个事件共用 |
| eventName | 事件名称 |
| kv | 事件的键值对 |
| action | 1，展示 2，点击 |
| newsId | 新闻Id |

### 1.4.2 用户后台上报日志

1537619705000| **{**

**"appkey"**: "browser"**,**

**"he"**: **{**

**"userId"**: "a12234sd"**,**

**"area"**: "BR"**,**

**"appVersion"**: "V1.0.2",

**"appTime"**: "1537533305000"

**},**

**"et"**: [**{**

**"eventName "**: "background"，

**"kv "**: **{**

**"createTime"**: "1537533305000"

**}**

**}]**

**}**

|  |  |
| --- | --- |
| 标签 | 含义 |
| 最前面的时间 | 服务端加上的时间 |
| eventName | background后台上报日志 |
| createTime | 后台活跃产生时间，即该后台启动时间 |

### 1.4.3 用户前台上报日志

1537619705000| **{**

**"appkey"**: "browser"**,**

**"he"**: **{**

**"userId"**: "a12234sd"**,**

**"area"**: "BR"**,**

**"appVersion"**: "V1.0.2",

**"appTime"**: "1537533305000"

**},**

**"et"**: [**{**

**"eventName "**: "foreground"，

**"kv "**: **{**

**"createTime"**: "1537533305000"

**}**

**}]**

**}**

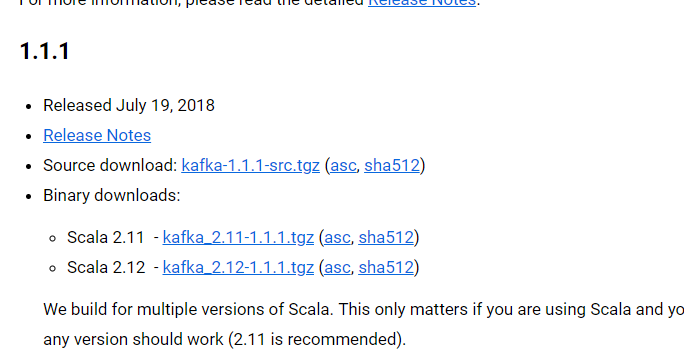
|  |  |
| --- | --- |
| 标签 | 含义 |
| 最前面的时间 | 服务端加上的时间 |
| eventName | foreground后台上报日志 |
| createTime | 前台活跃产生时间，即用户打开该应用的时候 |

## 1.5 产品选型

### 1.5.1 Kafka版本

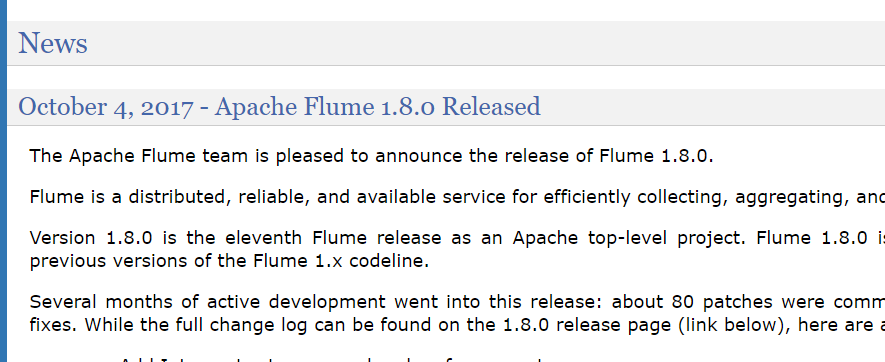
由于0.8的支持已经过时，所以我们选择1.1.1的kafka版本





### 1.5.2 Flume版本

选择1.8版本。而1.7版本只支持0.8的kafka



### 1.5.3 SpringCloud版本

Spring与后台大数据是解耦的，所以选最新的版本就可以了

### 1.5.4 产品选型列表

|  |  |
| --- | --- |
| 产品 | 版本 |
| Nginx | 1.12.2 |
| Hadoop | 2.7.2 |
| Spring Cloud | Finchley.SR1 |
| Spark | 2.3.1 |
| Flume | 1.8 |
| Kafka | 1.1.1 |
| Hive | 1.2.2 |
| Presto | presto-server-0.211, |
| Mongo | 4.0.2 |

# 第2章 预备知识

## 2.1 Flume

### 2.1.1 Flume 组件介绍

1. 1-8 Flume组件介绍

|  |  |
| --- | --- |
| 名称 | 解析 |
| Event | 一个数据单元，带有一个可选的消息头，其实就是一条消息，一个日志；（Kafka 的消息没有消息头，因此，Flume 的消息进入 Kafka 后，消息头就丢失了） |
| Flow | 数据流，Event 从源点到达目的点的迁移的抽象 |
| Client | 操作位于源点处的 Event，将其发送到 Flume Agent |
| Agent | 一个独立的 Flume 进程，包含组件 Source、Channel、Sink |
| Source | 用来获取 Event 并写入 Channel |
| Channel | 中转 Event 的一个临时存储，保存有 Source 组件传递过来的Event，可以认为是一个队列 |

### 2.1.2 Flume 组件选择

1. 多层 Flume

第一层 agent：

**Source**：TailDirSource

**Channel**：FileChannel

**Sink**：AvroSink

注意，TailDirSource 是 Flume 1.7 提供的 Source 组件，在 1.6 中并没有.

第二层 agent：

**Source**：AvroSource

**Channel**：FileChannel

**Sink**：KafkaSink

2. 单层 Flume

**Source**：TailDirSource

**Channel**：FileChannel

**Sink**：KafkaSink

### 2.1.3 Flume 采集系统组件解析

* 1. Source

1. Avro Source

侦听 Avro 端口并从外部 Avro 客户端流接收事件。当与另一个（上一跳）Flume 代理的内置 Avro Sink 配对时，它可以创建分层收集拓扑。

1. Taildir Source
2. Flume1.7 之前如果想要监控一个文件新增的内容，我们一般采用的 source 为 exec tail ,但是这会有一个弊端，就是当你的服务器宕机重启后，此时数据读取还是从头开始，这显然不是我们想看到的！在 Flume1.7 没有出来之前我们一般的解决思路为：当读取一条记录后，就把当前的记录的行号记录到一个文件中，宕机重启时，我们可以先从文件中获取到最后一次读取文件的行数，然后继续监控读取下去。保证数据不丢失、不重复。
3. Flume1.7 时新增了一个 source 的类型为 taildir，它可以监控一个目录下的多个文件，并且实现了实时读取记录保存的断点续传功能。

但是 Flume1.7 中如果文件重命名，那么会被当成新文件而被重新采集。

* 1. Channel

1. Memory Channel

Memory Channel 把 Event 保存在内存队列中，该队列能保存的 Event 数量有最大值上限。由于 Event 数据都保存在内存中，Memory Channel 有最好的性能，不过也有数据可能会丢失的风险，如果 Flume 崩溃或者重启，那么保存在 Channel 中的 Event 都会丢失。同时由于内存容量有限，当 Event 数量达到最大值或者内存达到容量上限，Memory Channel 会有数据丢失。

1. File Channel

File Channel 把 Event 保存在本地硬盘中，比 Memory Channel 提供更好的可靠性和可恢复性，不过要操作本地文件，性能要差一些。

1. Kafka Channel

Kafka Channel 把 Event 保存在 Kafka 集群中，能提供比 File Channel 更好的性能和比 Memory Channel 更高的可靠性。

* 1. Sink

1. Avro Sink

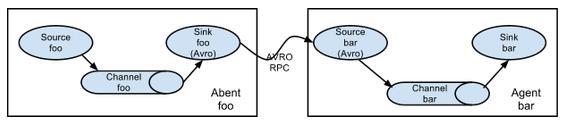
Avro Sink 是 Flume 的分层收集机制的重要组成部分。发送到此接收器的 Flume 事件变

1. Avro 事件，并发送到配置指定的主机名/端口对。事件将从配置的通道中按照批量配置的批量大小取出。
2. Kafka Sink

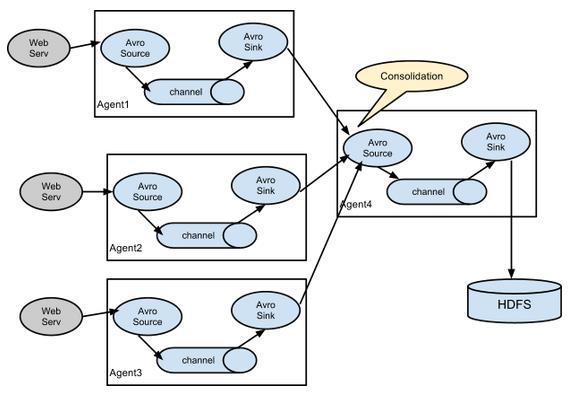
Kafka Sink 将会使用 FlumeEvent header 中的 topic 和 key 属性来将 event 发送给 Kafka。如果 FlumeEvent 的 header 中有 topic 属性，那么此 event 将会发送到 header 的 topic 属性指定的 topic 中。如果 FlumeEvent 的 header 中有 key 属性，此属性将会被用来对此 event 中的数据指定分区，具有相同 key 的 event 将会被划分到相同的分区中，如果 key 属性 null，那么 event 将会被发送到随机的分区中。

可以通过自定义拦截器来设置某个 event 的 header 中的 key 或者 topic 属性。

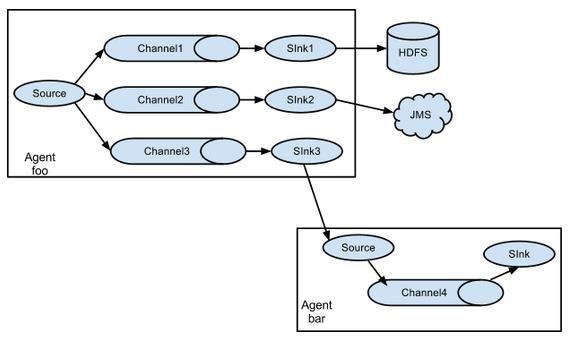
### 2.1.4 Flume 拓扑结构

****

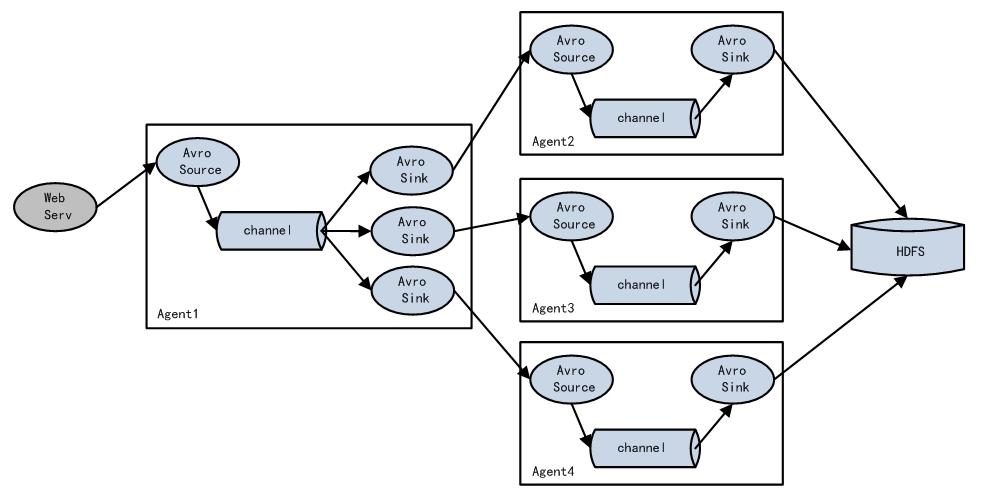
1. 2-1 Flume Agent 连接



1. 2-2 Flume Agent 聚合



1. 2-3 Flume 多路 Flow

****

1. 2-4 Flume 负载均衡

### 2.1.5 Flume Source,Channel处理器,拦截器和 Channel 选择之间的交互

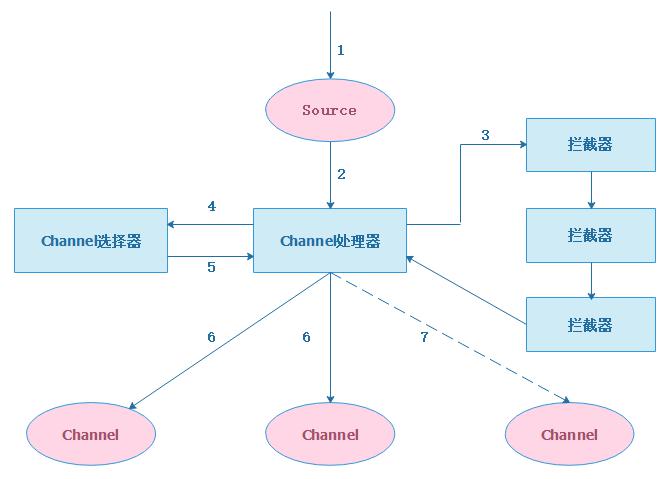
****

图 2-5 Flume 负载均衡

1. 接收事件；
2. 处理事件；
3. 将事件传递给拦截器链；
4. 将每个事件传递给 Channel 选择器；

1. 返回写入事件的 Channel 列表；
2. 将所有事件写入每个必需的 Channel，只有一个事务被打开；对于每个 Channel，所有事件都写为事务的一部分；
3. 利用可选 Channel 重复相同动作。

Flume 本身不限制 Agent 中 Source、Channel 和 Sink 的数量。因此 Flume Source 可以接收事件，并可以通过配置将事件复制到多个目的地。这使得 Source 通过 Channel 处理器、拦截器和 Channel 选择器，写入数据到 Channel 成为可能。

每个 Source 都有自己的 Channel 处理器。每次 Source 将数据写入 Channel，它是通过委派该任务到其 Channel 处理器来完成的。然后，Channel 处理器将这些事件传到一个或多个 Source 配置的拦截器中。

拦截器是一段代码，可以基于某些它完成的处理来读取时间和修改或删除时间。基于某些标准，如正则表达式，拦截器可以用来删除事件，为事件添加新报头或移除现有的报头等。每个 Source 可以配置成使用多个拦截器，按照配置中定义的顺序被调用，将拦截器的结果传递给链的下一个单元。这就是所谓的责任链的设计模式。一旦拦截器处理完事件，拦截器链返回的事件列表传递到 Channel 列表，即通过 Channel 选择器为每个事件选择的 Channel。

Source 可以通过处理器-拦截器-选择路由器写入多个 Channel。Channel 选择器是决定每个事件必须写入到 Source 附带的哪个 Channel 的组件。因此拦截器可以用来插入或删除事件中的数据，这样 Channel 选择器可以应用一些条件在这些事件上，来决定事件必须写入哪

1. Channel。Channel 选择器可以对事件应用任意过滤条件，来决定每个事件必须写入哪些

Channel，以及哪些 Channel 是必需的或可选的。

写入到必需的 Channel 失败将会导致 Channel 处理器抛出 ChannelException，表明 Source 必须重试该事件（实际上，所有的时间都在那个事务中），而未能写入可选 Channel 失败仅仅忽略它。一旦写出事件，处理器将会对 Source 指示成功状态，可能发送确认（ACK）给发送该事件的系统，并继续接受更多的事件。

1. 拦截器

拦截器（Interceptor）是简单插件式组件，设置在 Source 和 Source 写入数据的 Channel

之间。Source 接收到的事件在写入到对应的 Channel 之前，拦截器都可以转换或删除这些事件。每个拦截器实例只处理同一个 Source 接收到的事件。拦截器可以基于任意标准或转换事件，但是拦截必须返回尽可能多（或尽可能少）的事件，如同原始传递过来的事件。

多个拦截器组成一个有序的拦截器链。在一个链条中，可以添加任意数量的拦截器去转换从单个 Source 中来的事件。Source 将同一个事务的所有事件传递给 Channel 处理器，进而传递给拦截器链条，然后事件被传递给链条中的第一个拦截器。通过拦截器转换时间产生的一系列事件，传递到链条的下一个拦截器，以此类推。链条最后一个拦截器返回的最终事

件列表写入到 Channel 中。

因为拦截器必须在事件写入 Channel 之前完成转换操作，只有当拦截器已成功转换事件

后，RPC Source（和任何其他可能产生超时的 Source）才会响应发送事件的客户端或 Sink。

因此，在拦截器中进行大量重量级的处理并不是和一个好主意。如果拦截器中的处理时重量

级的、耗时的，那么需要相应地调整超时时间属性。

Flume 配置文件中，所有拦截器通用的唯一配置参数是 type 参数，改参数必须是拦截器的别名或者 Builder 类的完全限定类名（FQCN），该 Builder 类用于创建拦截器。正如前面提到的，可以设置任意数量的拦截器连接到单个的 Source。

拦截器是需要命名的组件，每个拦截器实例必须限定一个名字。为了给 Source 添加拦截器，需要列出 Source 应该连接的拦截器名字，这些拦截器就是 Source 应该连接到 Source 配置中 interceptors 参数的值代表的拦截器。原配置中以 interceptors. 开头的、后面跟着拦截器名称和参数的所有值都传递给拦截器。

1. Channel 选择器

Channel 选择器是决定 Source 接收的一个特定事件写入哪些 Channel 的组件。它们告知 Channel 处理器，然后将事件写入到每个 Channel。

由于 Flume 并不是两阶提交（不会等所有事件都写入成功后再一起提交，而是写一个提交一个），事件被写入到一个 Channel，然后在事件被写入到下一个 Channel 之前提交。如果写入一个 Channel 时出现故障，可能已经发生在其他 Channel 的相同事件的写入不能被

回滚。当这样的故障发生时，Channel 处理器抛出 ChannelException 并且事务失败。如果 Source 试图再次写入相同的事件（在大多数情况下，它会重试，只有类似 Syslog、Exec 等 Source 不能重试，因为没有办法再次生成相同的数据），重复的事件将写入到 Channel，而先前的提交实际上是成功的，这是在 Flume 管道发生重复的一种情况。

Channel 选择器配置是通过 Channel 处理器完成的，虽然配置看起来像 Source 子组件的配置。传递到 Channel 选择器的所有参数作为 Source 的上下文中的参数使用 selector 后缀传递。对于每个 Source，选择器通过使用一个配置参数 type 指定。Channel 选择器可以指定一

1. Channel 是必需的（required），另一组是可选的（optional）。

Flume 内置两种 Channel 选择器：replicating 和 mutiplexing。如果 Source 的配置中没有指定选择器，那么会自动使用复制 Channel 选择器。

replicating Channel 选择器复制每个事件到 Source 的 channels 参数所指定的所有 Channel 中。

multiplexing Channel 选择器是一种专门用于动态路由事件的 Channel 选择器，通过选择事件应该写入的 Channel，基于一个特定的事件头的值进行路由。

### 2.1.6 Flume Sink、Sink 运行器、Sink 组和 Sink 处理器

1. 2-6 Flume Sink、Sink 运行器、Sink 组合 Sink 处理器

Sink 运行器（Sink Runner）运行一个 Sink 组（Sink Group），Sink 组可以含有一个或多个 Sink。如果组中只存在一个 Sink，那么没有组将会更有效率。Sink 运行器仅仅是一个询问 Sink 组（或 Sink）来处理下一批事件的线程。每个 Sink 组有一个 Sink 处理器（Sink Processor），处理器选择组中的 Sink 之一去处理下一个事件集合。每个 Sink 只能从一个

Channel 获取数据（一个 Sink 只能有一个 Channel），尽管多个 Sink 可以从同一个 Channel 获取数据。选定的 Sink（或如果没有组，唯一的 Sink）从 Channel 中接受事件，并将事件写入到下一阶段或最终目的地。

1. Sink 组

Flume 配置框架为每个 Sink 组实例化一个 Sink 运行器，来运行 Sink 组。每个 Sink 组可以包含任意数量的 Sink。Sink 运行器持续请求 Sink 组，要求其中的一个 Sink 从自己的 Channel 中读取事件。Sink 组通常用于 RPC Sink，在层之间以负载均衡或故障转移方式发送

数据。

Sink 组中的每个 Sink 必须单独进行配置。这包括：Sink 从哪个 Channel 读取，写数据到哪些主机或者集群。在理想情况下，如果 Sink 组中建立了几个 Sink，所有的 Sink 将从相同的 Channel 读取，这将有利于在当前层以合理的速度清除数据，确保将要被发送到多台集群的数据，以一种支持负载均衡和故障转移的方式进行发送。

1. Load-Balancing Sink 处理器

Load-Balancing Sink 处理器从所有的 Sink 中选择一个 Sink，处理来自 Channel 的事件。

Sink 选择的顺序可以为 random 或者 round-robin。如果顺序被设置为 random，那么将随机从 Sink 组的 Sink 中选择一个，用来从自己的 Channel 中移除事件并将它们写出。round-robin选项使 Sink 以循环的方式被选择：每个选择循环调用定义 Sink 组中指定顺序 Sink 的 process

方法。如果 Sink 写入到一个失败的 Agent 或者速度太慢的 Agent，会导致超时，Sink 处理器会选择另一个 Sink 写数据。

Sink 处理器可以配置将失败的 Sink 加入黑名单，回退时间以指数方式增长直到达到上限值。这能确保相同的 Sink 不会循环重复尝试且不浪费资源，直到回退时间过期。

### 2.1.7 Flume Taildir Source 移植

如果使用 0.8 版本 Kafka 并配套 1.6 版本 Flume，由于 Flume 1.6 版本没有 Taildir Source 组件，因此，需要将 Flume 1.7 中的 Taildir Source 组件源码编译打包后，放入 Flume1.6 安装目录的 lib 文件目录下。

所谓移植，就是将 Flume1.7 版本中 Taildir Source 的相关文件全部提取出来，然后将这些文件放入新建的项目中进行编译打包，将打包出的 jar 包放入 Flume1.6 安装目录的 lib 目录下即可。

在本项目中，已经将 Taildir Source 的源码放入了 taildirsource 项目中，直接编译项目，打包后放入 Flume1.6 安装目录的 lib 文件目录下即可。

1. Flume 配置文件中指定全类名即可使用 Taildir Source 组件。

a1.sources.r1.type = com.atguigu.flume.source.TaildirSource

如果使用 0.10 版本 Kafka 并配套 1.7 及以上版本的 Flume，那么无需手动移植，1.7 及以上版本的 Flume 自带 Taildir Source。

Taildir Source 有三个坑：不支持 Windows，只支持 UTF-8，文件重命名后重新读取。

### 2.1.8 Flume Taildir Source 源码修改

TaildirSource 中维护着两个核心结构，一个是记录着所有监控文件相关信息的 Map 结构，一个是实时记录每个文件读取位置的 position file。

1. TaildirSource 的源码中，维护了一个<inode，TailFile>为元素的 Map，用来记录指定目录下的所有采集文件，当有新文件时，会写入此 Map 中，因此，所有被监控文件的信息都集中记录在这一 Map 结构中，当有新文件时，此 Map 中会产生新的 K-V 对。

（Linux 中储存文件元数据的区域就叫做 inode，每个 inode 都有一个号码，操作系统用 inode 号码来识别不同的文件，Unix/Linux 系统内部不使用文件名，而使用 inode 号码来识别文件，当文件名称改变时，inode 并不会改变）

Map 的 key 为 inode，value 如下：

private RandomAccessFile raf;

private final String path;

private final long inode;

private long pos;

private long lastUpdated;

private boolean needTail;

private final Map<String, String> headers;

【更多 Java、HTML5、Android、python、大数据 资料下载，可访问尚硅谷（中国）官网下载区】

private byte[] buffer;

private byte[] oldBuffer;

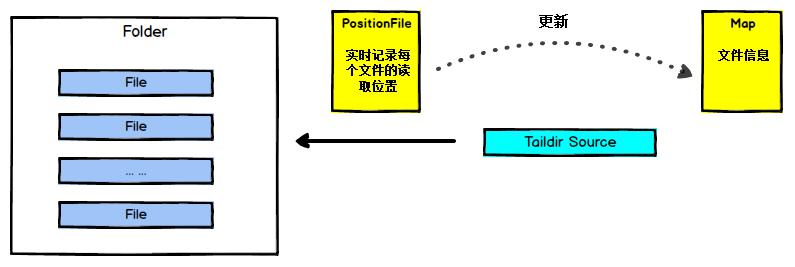
private int bufferPos;

private long lineReadPos;

可以看到 value 中记录了文件路径、inode、读取位置 pos、更新时间等信息。

在维护上述的 Map 结构时，判断文件是否为新文件时加入了 inode 和文件名的双重判断，这使得一旦文件重命名，将会被判定为新文件，因此此处需要修改 updateTailFiles()。

与此同时，TaildirSource 在运行过程中维护了一个 position file，对于文件的读取位置会实时记录到 position file 中，包括 inode、pos 和 path，然后根据 position file 中的内容更新 Map 中的数据。



1. TaildirSource 的源码中，通过 inode 和文件名来进行唯一文件的判定，也就是说，一旦文件改名，那么 TaildirSource 会将它当成新文件而重新读取，这是不合理的，会造成数据的重复读取。

为了解决这个问题，我们对 ReliableTaildirEventReader.java 文件进行修改，具体是对此文件中的 updateTailFiles 函数和 loadPositionFile 函数进行修改。

1. updateTailFiles()

UpdateTailFiles 函数扫描指定的监控目录是否产生了新文件或者文件是否被追加了内容。

UpdateTailFiles 函数使用了一个 Map 结构保存所有被监视文件的信息，对 Map 的每个 TailFile 中的 pos 进行更新时，传入的是 position file 中的旧文件名，而在 updateTailFiles()中已经修改了源码，修改后重命名会覆盖 Map 中的原始文件，TailFile 中的信息同步更新，不会再将重命名的文件当成新文件，而原始的 loadPositionFile()会将 json 中的旧文件名传入

updatePos()，如果不修改会导致名称不匹配而更新 pos 失败，因此也应该对 loadPositionFile() 进行修改。

代码清单 3-4 updateTailFiles()

/\*\*

1. Update tailFiles mapping if a new file is created or appends are detected
2. to the existing file.
3. 扫描指定的监控目录是否产生了新文件或者文件是否被追加了内容

\*/

public List<Long> updateTailFiles (boolean skipToEnd) throws IOException { updateTime = System.currentTimeMillis();

List<Long> updatedInodes = Lists.newArrayList();

【更多 Java、HTML5、Android、python、大数据 资料下载，可访问尚硅谷（中国）官网下载区】

for (TaildirMatcher taildir : taildirCache) {

Map<String, String> headers = headerTable.row(taildir.getFileGroup());

//遍历所有匹配的文件

for (File f : taildir.getMatchingFiles()) {

//得到本地文件的 inode(储存文件元信息的区域就叫做 inode，inode 包含除了文件名以外的所有文件信息) //文件由唯一的 inode，不论文件是否重命名，inode 不变

long inode = getInode(f);

//tailFiles 是一个 Map，以 inode 为 key，以 TailFile 为 value //第一次遍历，此 inode 对应的 Map 项肯定不存在

TailFile tf = tailFiles.get(inode);

//源码中导致文件重命名后被重新读取的罪魁祸首

1. 当文件重命名后， !tf.getPath().equals(f.getAbsolutePath())为 True，那么就会创建新的TailFile，然后覆盖 Map 中原有的 key-value 对

//if (tf == null || !tf.getPath().equals(f.getAbsolutePath())) { if (tf == null) {

//如果 Map 中对应文件为空，那么就创建一个 TailFile 对象

//skipToEnd 可配置，决定是否从文件开始位置读取数据还是直接跳到文件结尾 long startPos = skipToEnd ? f.length() : 0;

//openFile 中根据传入的参数 new 了一个新的 TailFile tf = openFile(f, headers, inode, startPos);

} else { //不为空时进入

//如果文件重命名则进入此分支，由于是对于源码的修改导致重命名后进入，必须再次修改源码以处理重命名

情况

//判断此文件的更新时间是否比 Map 中存储的文件更新时间要新

boolean updated = tf.getLastUpdated() < f.lastModified(); if (updated) {

//如果 Map 含有对应项，但是得到的 tf 中封装的文件为 null，需要重新创建 tf if (tf.getRaf() == null) {

tf = openFile(f, headers, inode, tf.getPos());

}

* 1. 如果 Map 中记录的读取位置 Pos 已经超过了文件长度，那么设置 Map 中的 Pos 值为 0，即重新从 0 开

始

if (f.length() < tf.getPos()) {

logger.info("Pos " + tf.getPos() + " is larger than file size! "

1. "Restarting from pos 0, file: " + tf.getPath() + ", inode: " + inode); tf.updatePos(tf.getPath(), inode, 0);

}

}

//重命名后，Map 中的文件名还是老的文件名，因此使用 openFIle 重新创建 TailFile 用来替换原数据 //modify by zhangpeng

if (!tf.getPath().equals(f.getAbsolutePath())) { tf = openFile(f, headers, inode, tf.getPos());

}

//modify by zhangpeng end

tf.setNeedTail(updated);

}

//将 inode 及其对应的 tf 加入 Map 中

tailFiles.put(inode, tf);

updatedInodes.add(inode);

}

}

return updatedInodes;

}

2. loadPositionFile ()

根据 position file 更新 TailFile Map 中每个文件的 pos：

代码清单 3-5 loadPositionFile()

/\*\*

1. Load a position file which has the last read position of each file.
2. 加载并解析记录了每个文件最新读取位置的 position file
3. If the position file exists, update tailFiles mapping.
4. 如果 position file 存在则更新 tailFiles 映射

\*/

public void loadPositionFile(String filePath) {

Long inode, pos;

String path;

FileReader fr = null;

JsonReader jr = null;

//对 position file 进行读取和解析

try {

fr = new FileReader(filePath);

jr = new JsonReader(fr);

jr.beginArray();

while (jr.hasNext()) {

inode = null;

pos = null;

path = null;

jr.beginObject();

while (jr.hasNext()) {

switch (jr.nextName()) {

case "inode":

inode = jr.nextLong();

break;

case "pos":

pos = jr.nextLong();

break;

case "file":

path = jr.nextString();

break;

}

}

jr.endObject();

for (Object v : Arrays.asList(inode, pos, path)) { Preconditions.checkNotNull(v, "Detected missing value in position file. "

+ "inode: " + inode + ", pos: " + pos + ", path: " + path);

}

//判断 position file 中的 inode 是否存在于 TailFile Map 中 TailFile tf = tailFiles.get(inode);

//根据对 updatePos 的分析，当出现重命名时，position file 中的 path 项对应的文件名是旧文件名，而通过 updateTailFiles()已经将 Map 中的文件名更新成了重命名后的文件名

//因此，为了 updatePos 能够顺利更新 pos，应该传入 tf.getPath()，即新文件名，tailfile 与 tailfile

自身的文件名的比较必然是相等的

//modify by zhangpeng

//if (tf != null && tf.updatePos(path, inode, pos)) {

if (tf != null && tf.updatePos(tf.getPath(), inode, pos)) { tailFiles.put(inode, tf);

} else {

logger.info("Missing file: " + path + ", inode: " + inode + ", pos: " + pos);

}

}

jr.endArray();

} catch (FileNotFoundException e) {

logger.info("File not found: " + filePath + ", not updating position"); } catch (IOException e) {

logger.error("Failed loading positionFile: " + filePath, e);

} finally { try {

if (fr != null) fr.close(); if (jr != null) jr.close();

} catch (IOException e) {

logger.error("Error: " + e.getMessage(), e);

}

}

}

代码清单 3-6 loadPositionFile()

public boolean updatePos(String path, long inode, long pos) throws IOException {

1. 注意 updatePos 判断是否更新读取位置时检查的是 inode 和文件名，如果重命名而传入的仍是 position file 中记录的原来的名称，那么此处将不会更新 pos

if (this.inode == inode && this.path.equals(path)) {

setPos(pos);

updateFilePos(pos);

logger.info("Updated position, file: " + path + ", inode: " + inode + ", pos: " + pos);

return true;

}

return false;

}

### 2.1.9 Flume 的停止

使用 kill 停止 Flume 进程。

不可使用 kill -9，因为 Flume 内部注册了很多钩子函数执行善后工作，如果使用 kill -9 会导致钩子函数不执行，使用 kill 时，Flume 内部进程会监控到用户的操作，然后调用钩子函数，执行一些善后操作，正常退出。

### 2.1.10 Flume 数据丢失问题的讨论

在一些网络资料中提出当 Flume 的数据量达到 70MB/s 以上时，就会出现丢失数据的情况，但是根据 Flume 的架构原理，Flume 是不可能丢失数据的，其内部有完善的事务机制，

Source 到 Channel 是事务性的，Channel 到 Sink 是事务性的，因此这两个环节不会出现数据的丢失，唯一可能丢失数据的情况是 Channel 采用 memoryChannel，agent 宕机导致数据丢

失，或者 Channel 存储数据已满，导致 Source 不再写入，未写入的数据丢失。并且，在实际的项目开发和运行过程中，并没有出现过 Flume 丢失数据的情况（以滴滴为例），因此， Flume 在数据量大的时候丢失数据的论断还有待商榷。

Flume 不会丢失数据，但是有可能造成数据的重复，例如数据已经成功由 Sink 发出，但是没有接收到响应，Sink 会再次发送数据，此时可能会导致数据的重复。

## 2.2 Kafka

(同硅谷大数据项目之手机APP信息统计分析项目)

## 2.3 Hive/ Spark SQL

Hive: (同硅谷大数据项目之手机APP信息统计分析项目)

## 2.4 Java & Maven

Apache Maven是一个软件项目管理和综合工具。基于项目对象模型（POM）的概念，Maven可以从一个中心资料片管理项目构建，报告和文件。

## 2.5 Spark Core

(同硅谷大数据项目之手机APP信息统计分析项目)

## 2.6 Spark Streaming

(同硅谷大数据项目之手机APP信息统计分析项目)

## 2.7 Crontab 任务调度

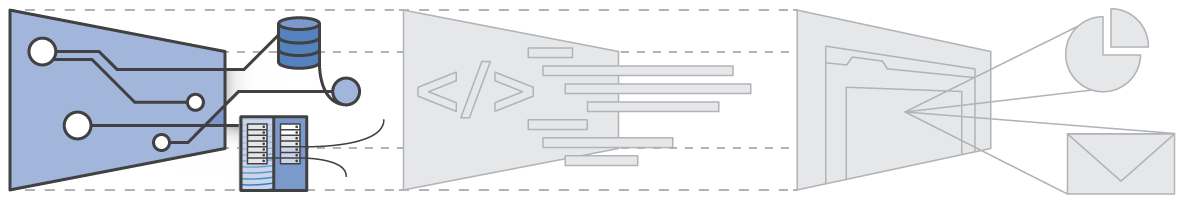
(同硅谷大数据项目之手机APP信息统计分析项目)

## 2.8 云计算

**云计算模型**

云计算的模型主要有三种。每种模型代表着云计算世界的一个独特部分。

### 基础设施即服务 (IaaS)

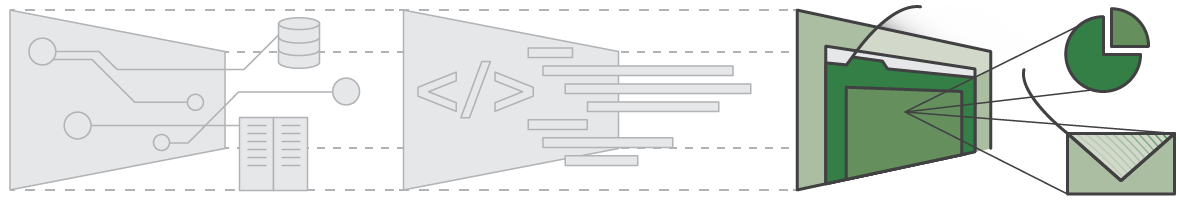


基础设施即服务有时缩写为 IaaS，包含云 IT 的基本构建块，通常提供对联网功能、计算机（虚拟或专用硬件）以及数据存储空间的访问。基础设施即服务提供最高等级的灵活性和对 IT 资源的管理控制，其机制与现今众多 IT 部门和开发人员所熟悉的现有 IT 资源最为接近。

### 平台即服务 (PaaS) 平台即服务

平台即服务消除了组织对底层基础设施（一般是硬件和操作系统）的管理需要，让您可以将更多精力放在应用程序的部署和管理上面。这有助于提高效率，因为您不用操心资源购置、容量规划、软件维护、补丁安装或任何与应用程序运行有关的不能产生价值的繁重工作。

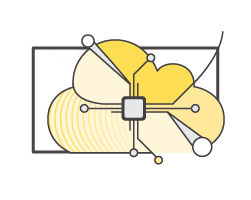
### 软件即服务 (SaaS)



软件即服务提供一种完善的产品，其运行和管理皆由服务提供商负责。通常人们所说的软件即服务指的是终端用户应用程序。使用 SaaS 产品时，服务的维护和底层基础设施的管理都不用您操心，您只需要考虑怎样使用 SaaS 软件就可以了。SaaS 的常见应用是基于 Web 的电子邮件，在这种应用场景中，您可以收发电子邮件而不用管理电子邮件产品的功能添加，也不需要维护电子邮件程序所运行的服务器和操作系统。

云计算部署模型

### [云](https://aws.amazon.com/cn/what-is-cloud-computing/)

[](https://aws.amazon.com/cn/what-is-cloud-computing/)

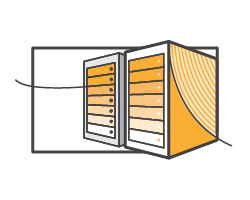
基于云的应用程序完全部署在云中且应用程序的所有组件都在云中运行。云中的应用程序分为两种，一种是在云中创建，另一种是从现有基础设施迁移到云中以利用云计算的优势。基于云的应用程序可以构建在基础设施组件上，也可以使用较高级的服务，这些服务提供了从核心基础设施的管理、架构和扩展要求中抽象提取的能力。

### 混合

[](https://aws.amazon.com/cn/hybrid/)

混合部署是一种在基于云的资源和非云现有资源之间连接基础设施和应用程序的方法。混合部署最常见的方法是在云和现有内部基础设施之间将组织的基础设施扩展到云中，同时将云资源与内部系统进行连接。

### 内部



使用虚拟化和资源管理工具在内部部署资源往往被称作“私有云”。内部部署无法提供云计算的诸多优势，但有时采用这种方案是为了能够提供专用资源。大多数情况下，这种部署模型与旧式 IT 基础设施无异，都通过应用程序管理和虚拟化技术尽可能提高资源利用率。

目前主流的云计算服务供应商有

* 亚马逊云
* 微软云
* 谷歌云
* 阿里云
* 腾讯云

AWS中云产品中的服务器为EC2，硬盘存储为EBS，分布式文件系统为S3

## 2.9 大数据仓库

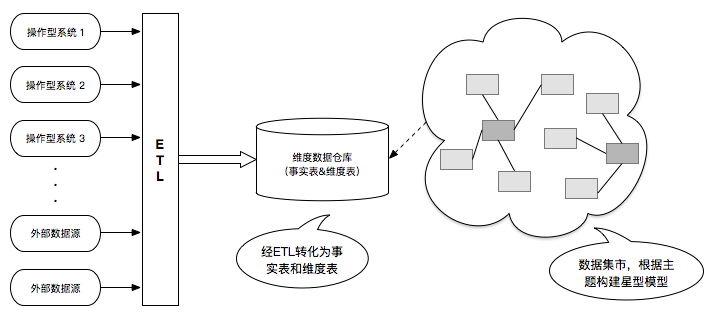
数据建模的概念：

### 2.9.1 kimball建模

1）Kimball 的模型的数据源往往是给定的若干个数据库表，数据较为稳定但是数据之间的关联关系比较复杂，需要从这些OLTP中产生的事务型数据结构抽取出分析型数据结构。Kimball 是以最终任务为导向，将数据按照目标拆分出不同的表需求，通过ETL导入数据集市层

2）**Kimball模型将分散异构的数据源经ETL转化为事实表和维度表导入数据集市**，数据集市由若干个事实表和维度表组成

3）在数据集市将事实表和维度表根据分析主题组合后导入数据仓库中，用于数据分析



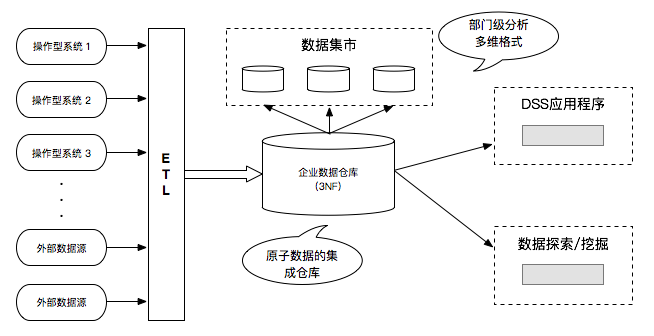
### 2.9.2 Inmon建模

1）操作型系统的数据和体系外数据需要经过ETL过程，加载到企业数据仓库中

2）企业数据仓库是企业信息化工厂的枢纽，是原子数据的集成仓库，其目的是将附加的数据存储用于各类分析型系统；在数据仓库中会对数据进行清洗，并抽取实体-关系。

3）数据集市是针对不同主题的聚集区域

基本上就是将数据库的数据一对一导入进数据仓库中。要满足数据库的三大范式（3NF）



### 2.9.3 逻辑层架构

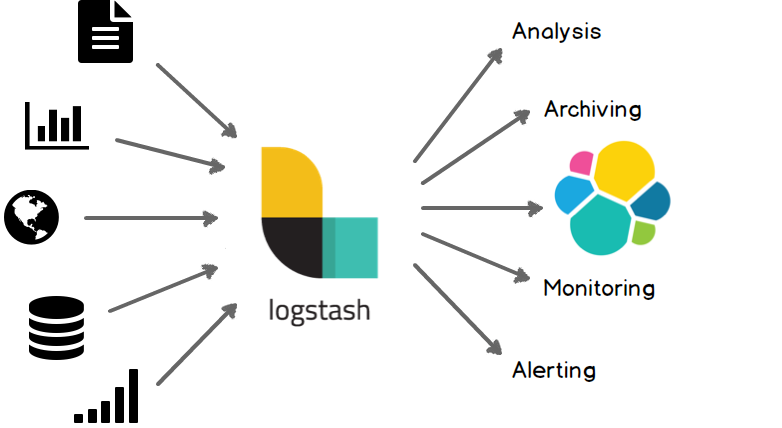
**分为：**

**STAGE**接口信息模型、ODS/DWD细节数据层，MID轻度综合层、DM数据集市、元数据信息模型。

|  |  |  |
| --- | --- | --- |
| 数据模型规范要求的层次 | 数据中心物理模型层次名称 | 物理表前缀（参考） |
|  | 维度数据层 | DIM\_ |
| 数据集市层(DM) | 展示层 | DM\_ |
| 主题域 | DW\_F\_ |
| 轻度汇总层(MID) | 轻度汇总层 | DW\_M\_ |
| 细节数据层（ODS/DWD） | 细节数据层 | DW\_V\_ |
| 接口层（STAGE） | 接口层 | SRC\_ |

## 2.10 Logstash

### 简介



logstash是一个实时流水式开源数据收集引擎。具有强大的plugin。可以根据自己的业务场景选择不同的input filter output。绝大多数情况下都是结合ElasticSearch Kibana一起使用的，俗称ELK。

### 模块介绍

Logstash使用管道方式进行日志的搜集处理和输出。有点类似\*NIX系统的管道命令 xxx | ccc | ddd，xxx执行完了会执行ccc，然后执行ddd。

在logstash中，包括了三个阶段:

输入input –> 处理filter（不是必须的） –> 输出output

### 配置文件说明

前面介绍过logstash基本上由三部分组成，input、output以及用户需要才添加的filter，因此标准的配置文件格式如下：

input { } filter { } output { }

### 执行说明

|  |  |  |
| --- | --- | --- |
| 1 |  | bin/logstash -f demo.conf |

## 2.11 MySQL(RDS)

### 关系型数据库

    关系型数据库，是指采用了关系模型来组织数据的数据库。  
    关系模型是在1970年由IBM的研究员E.F.Codd博士首先提出的，在之后的几十年中，关系模型的概念得到了充分的发展并逐渐成为主流数据库结构的主流模型。  
    简单来说，关系模型指的就是二维表格模型，而一个关系型数据库就是由二维表及其之间的联系所组成的一个数据组织。  
**关系模型中常用的概念：  
    关系**：可以理解为一张二维表，每个关系都具有一个关系名，就是通常说的表名  
    **元组**：可以理解为二维表中的一行，在数据库中经常被称为记录  
    **属性**：可以理解为二维表中的一列，在数据库中经常被称为字段  
    **域**：属性的取值范围，也就是数据库中某一列的取值限制  
    **关键字**：一组可以唯一标识元组的属性，数据库中常称为主键，由一个或多个列组成  
    **关系模式**：指对关系的描述。其格式为：关系名(属性1，属性2， ... ... ，属性N)，在数据库中成为表结构  
  
关系型数据库的优点：  
    容易理解：二维表结构是非常贴近逻辑世界的一个概念，关系模型相对网状、层次等其他模型来说更容易理解  
    使用方便：通用的SQL语言使得操作关系型数据库非常方便  
    易于维护：丰富的完整性(实体完整性、参照完整性和用户定义的完整性)大大减低了数据冗余和数据不一致的概率  
  
**关系型数据库瓶颈**  
  1).高并发读写需求  
    网站的用户并发性非常高，往往达到每秒上万次读写请求，对于传统关系型数据库来说，硬盘I/O是一个很大的瓶颈  
  2).海量数据的高效率读写  
    网站每天产生的数据量是巨大的，对于关系型数据库来说，在一张包含海量数据的表中查询，效率是非常低的  
  3).高扩展性和可用性  
    在基于web的结构当中，数据库是最难进行横向扩展的，当一个应用系统的用户量和访问量与日俱增的时候，数据库却没有办法像web server和app server那样简单的通过添加更多的硬件和服务节点来扩展性能和负载能力。对于很多需要提供24小时不间断服务的网站来说，对数据库系统进行升级和扩展是非常痛苦的事情，往往需要停机维护和数据迁移。  
  
**对网站来说，关系型数据库的很多特性不再需要了**：  
  事务一致性: 关系型数据库在对事物一致性的维护中有很大的开销，而现在很多web2.0系统对事物的读写一致性都不高  
  读写实时性: 对关系数据库来说，插入一条数据之后立刻查询，是肯定可以读出这条数据的，但是对于很多web应用来说，并不要求这么高的实时性，比如发一条消息之后，过几秒乃至十几秒之后才看到这条动态是完全可以接受的  
  复杂SQL，特别是多表关联查询: 任何大数据量的web系统，都非常忌讳多个大表的关联查询，以及复杂的数据分析类型的复杂SQL报表查询，特别是SNS类型的网站（SNS，专指社交网络服务，包括了社交软件和社交网站。），从需求以及产品阶级角度，就避免了这种情况的产生。往往更多的只是单表的主键查询，以及单表的简单条件分页查询，SQL的功能极大的弱化了  
  
    在关系型数据库中，导致性能欠佳的最主要原因是多表的关联查询，以及复杂的数据分析类型的复杂SQL报表查询。为了保证数据库的ACID特性，我们必须尽量按照其要求的范式进行设计，关系型数据库中的表都是存储一个格式化的数据结构。每个元组字段的组成都是一样，即使不是每个元组都需要所有的字段，但数据库会为每个元组分配所有的字段，这样的结构可以便于标语表之间进行链接等操作，但从另一个角度来说它也是关系型数据库性能瓶颈的一个因素。

### NoSQL

    NoSQL一词首先是Carlo Strozzi在1998年提出来的，指的是他开发的一个没有SQL功能，轻量级的，开源的关系型数据库。这个定义跟我们现在对NoSQL的定义有很大的区别，它确确实实字如其名，指的就是“没有SQL”的数据库。但是NoSQL的发展慢慢偏离了初衷，我们要的不是“no sql”，而是“no relational”，也就是我们现在常说的非关系型数据库了。  
    2009年初，Johan Oskarsson举办了一场关于开源分布式数据库的讨论，Eric Evans在这次讨论中再次提出了NoSQL一词，用于指代那些非关系型的，分布式的，且一般不保证遵循ACID原则的数据存储系统。Eric Evans使用NoSQL这个词，并不是因为字面上的“没有SQL”的意思，他只是觉得很多经典的关系型数据库名字都叫“\*\*SQL”,所以为了表示跟这些关系型数据库在定位上的截然不同，就是用了“NoSQL“一词。  
注：数据库事务必须具备ACID特性，ACID是Atomic原子性，Consistency一致性，Isolation隔离性，Durability持久性。  
    非关系型数据库提出另一种理念，例如，以键值对存储，且结构不固定，每一个元组可以有不一样的字段，每个元组可以根据需要增加一些自己的键值对，这样就不会局限于固定的结构，可以减少一些时间和空间的开销。使用这种方式，用户可以根据需要去添加自己需要的字段，这样，为了获取用户的不同信息，不需要像关系型数据库中，要对多表进行关联查询。仅需要根据id取出相应的value就可以完成查询。但非关系型数据库由于很少的约束，他也不能够提供像SQL所提供的where这种对于字段属性值情况的查询。并且难以体现设计的完整性。他只适合存储一些较为简单的数据，对于需要进行较复杂查询的数据，SQL数据库显的更为合适。  
  
**2-1.非关系型数据库分类**  
    由于非关系型数据库本身天然的多样性，以及出现的时间较短，因此，不想关系型数据库，有几种数据库能够一统江山，非关系型数据库非常多，并且大部分都是开源的。  
    这些数据库中，其实实现大部分都比较简单，除了一些共性外，很大一部分都是针对某些特定的应用需求出现的，因此，对于该类应用，具有极高的性能。依据结构化方法以及应用场合的不同，主要分为以下几类：  
    **1).面向高性能并发读写的key-value数据库**：key-value数据库的主要特点即使具有极高的并发读写性能，Redis,Tokyo Cabinet,Flare就是这类的代表  
    **2).面向海量数据访问的面向文档数据库**：这类数据库的特点是，可以在海量的数据中快速的查询数据，典型代表为MongoDB以及CouchDB  
    **3).面向可扩展性的分布式数据库**：这类数据库想解决的问题就是传统数据库存在可扩展性上的缺陷，这类数据库可以适应数据量的增加以及数据结构的变化

### 关系型数据库 V.S. 非关系型数据库

    关系型数据库的最大特点就是事务的一致性：传统的关系型数据库读写操作都是事务的，具有ACID的特点，这个特性使得关系型数据库可以用于几乎所有对一致性有要求的系统中，如典型的银行系统。  
    但是，在网页应用中，尤其是SNS应用中，一致性却不是显得那么重要，用户A看到的内容和用户B看到同一用户C内容更新不一致是可以容忍的，或者说，两个人看到同一好友的数据更新的时间差那么几秒是可以容忍的，因此，关系型数据库的最大特点在这里已经无用武之地，起码不是那么重要了。  
    相反地，关系型数据库为了维护一致性所付出的巨大代价就是其读写性能比较差，而像微博、facebook这类SNS的应用，对并发读写能力要求极高，关系型数据库已经无法应付(在读方面，传统上为了克服关系型数据库缺陷，提高性能，都是增加一级memcache来静态化网页，而在SNS中，变化太快，memchache已经无能为力了)，因此，必须用新的一种数据结构存储来代替关系数据库。  
    关系数据库的另一个特点就是其具有固定的表结构，因此，其扩展性极差，而在SNS中，系统的升级，功能的增加，往往意味着数据结构巨大变动，这一点关系型数据库也难以应付，需要新的结构化数据存储。  
    于是，非关系型数据库应运而生，由于不可能用一种数据结构化存储应付所有的新的需求，因此，非关系型数据库严格上不是一种数据库，应该是一种数据结构化存储方法的集合。

    必须强调的是，数据的持久存储，尤其是海量数据的持久存储，还是需要一种关系数据库这员老将。

## 2.12 MongoDB

MongoDB 是由C++语言编写的，是一个基于分布式文件存储的开源数据库系统。

在高负载的情况下，添加更多的节点，可以保证服务器性能。

MongoDB 旨在为WEB应用提供可扩展的高性能数据存储解决方案。

MongoDB 将数据存储为一个文档，数据结构由键值(key=>value)对组成。MongoDB 文档类似于 JSON 对象。字段值可以包含其他文档，数组及文档数组

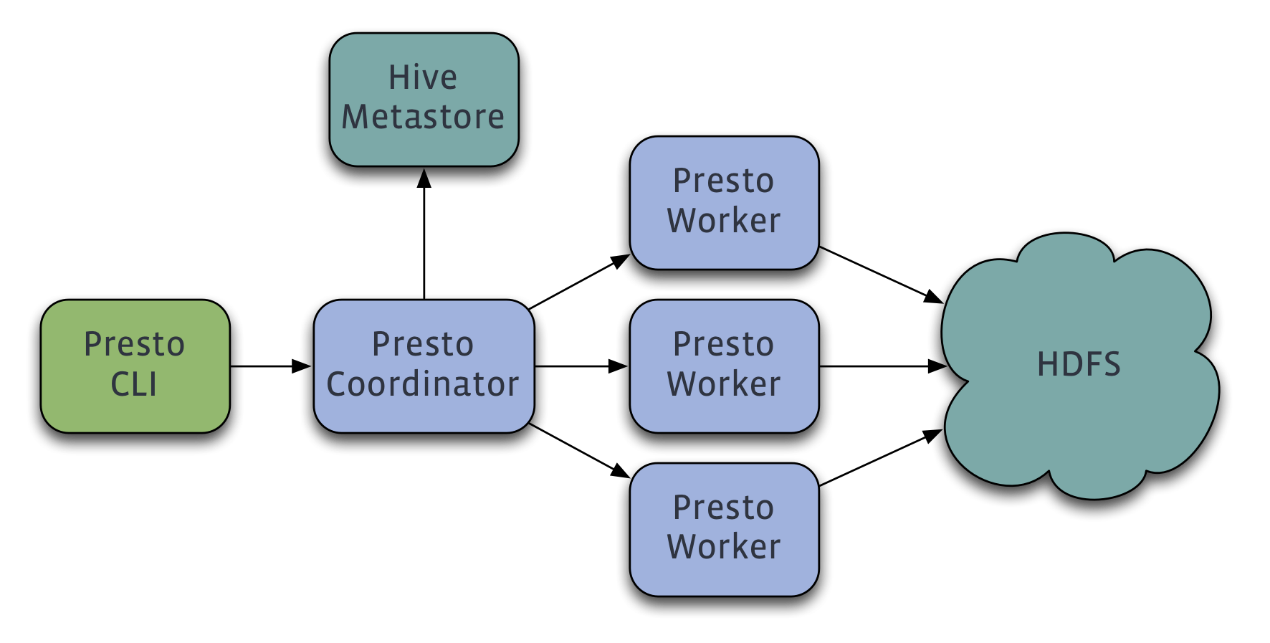


Mongo属于NoSQL数据库

## 2.13 Presto

Presto是一个分布式sql查询引擎，支持响应式查询，多数据源，大数据数据量。它源于facebook。

包含coordinator和多个worker。Coordinator解析sql，变成查询计划，然后分发给worker进行处理。它具有低延时



## 2.14 SpringBoot

使用Spring Boot可以让我们快速创建一个基于Spring的项目，而让这个Spring项目跑起来我们只需要很少的配置就可以了。Spring Boot主要有如下核心功能：

1.独立运行的Spring项目

Spring Boot可以以jar包的形式来运行，运行一个Spring Boot项目我们只需要通过java -jar xx.jar类运行。非常方便。

2.内嵌Servlet容器

Spring Boot可以内嵌Tomcat，这样我们无需以war包的形式部署项目。

3.提供starter简化Maven配置

使用Spring或者SpringMVC我们需要添加大量的依赖，而这些依赖很多都是固定的，这里Spring Boot 通过starter能够帮助我们简化Maven配置。

4.自动配置Spring

5.准生产的应用监控

6.无代码生成和xml配置

## 2.15 SLF4J

SLF4J的全称是Simple Logging Facade for Java，即简单日志门面。SLF4J并不是具体的日志框架，而是作为一个简单门面服务于各类日志框架，如java.util.logging, logback和log4j。

SLF4J提供了统一的记录日志的接口，对不同日志系统的具体实现进行了抽象化，只要按照其提供的方法记录即可，最终日志的格式、记录级别、输出方式等通过绑定具体的日志系统来实现。

使用SLF4J的好处在于，你只需要按统一的方式写记录日志的代码，如：

public class LoggerTest {

private static final Logger logger = LoggerFactory.getLogger(Tester.class);

public static void main(String[] args) {

logger.info("Current Time: {}", System.currentTimeMillis());

}

}

*SLF4J支持{}作为占位符，等价于C语言中的%s，而不必再进行字符串的拼接，效率有显著的提升（见后面运行结果）。*

而无需关心日志是通过哪个日志系统，以什么风格输出的。因为它们取决于部署项目时绑定的日志系统。   
例如，在项目中使用了SLF4J记录日志，并且绑定了log4j，则日志会以log4j的风格输出；后期需要改为以logback的风格输出日志，只需要将log4j替换成logback即可，不用修改项目中的代码

# 第3章 项目配置

## 3.1 Nginx

### 3.1.1 Nginx 安装

注意：以下操作均使用 root 用户。

* 1. 安装 PCRE

1. 下载 PCRE 安装包

wget http://downloads.sourceforge.net/project/pcre/pcre/8.35/pcre-8.35.tar.gz

1. 解压安装包

tar zxvf pcre-8.35.tar.gz

1. 进入安装包目录

cd pcre-8.35

1. 编译安装

./configure

如果编译过程中报错：error: You need a C++ compiler for C++ support，这是由于没有安装 gcc，输入以下指令进行安装：

yum install -y gcc gcc-c++

安装 pcre：

make && make install

1. 查看 pcre 版本

pcregrep -V

* 1. 安装 Nginx

1. 解压安装包

tar zxvf nginx-1.12.2.tar.gz

1. 进入安装包目录

cd nginx-1.12.2

1. 编译安装

./configure --prefix=/usr/local/webserver/nginx --with-http\_stub\_status\_module

--with-http\_ssl\_module --with-pcre=/usr/local/src/pcre-8.35

make && make install

如果报出以下错误，需要安装 OpenSSL

./configure: error: SSL modules require the OpenSSL library.

安装指令如下：

yum -y install openssl openssl-devel

1. 查看 Nginx 版本

/usr/local/webserver/nginx/sbin/nginx -v

### 3.1.2 Nginx 负载均衡配置

代码清单 3-1 Nginx负载均衡配置

#user nobody;

worker\_processes 1;

#error\_log logs/error.log;

#error\_log logs/error.log notice;

#error\_log logs/error.log info;

#pid logs/nginx.pid;

events {

worker\_connections 1024;

}

http {

include mime.types;

default\_type application/octet-stream;

#log\_format main '$remote\_addr - $remote\_user [$time\_local] "$request" '

1. '$status $body\_bytes\_sent "$http\_referer" '
2. '"$http\_user\_agent" "$http\_x\_forwarded\_for"';

#access\_log logs/access.log main;

sendfile on;

#tcp\_nopush on;

#keepalive\_timeout 0;

keepalive\_timeout 65;

#服务器的集群

upstream netitcast.com {

#服务器集群名字

#服务器配置 weight 是权重的意思，权重越大，分配的概率越大。

#server 127.0.0.1:18080;

#server 127.0.0.1:28080;

server 192.168.10.200:18080;

server 192.168.10.200:28080;

}

server {

listen 80;

server\_name localhost;

location / {

proxy\_pass http://netitcast.com;

proxy\_redirect default;

}

#error\_page 404 /404.html;

1. redirect server error pages to the static page /50x.html

error\_page 500 502 503 504 /50x.html; location = /50x.html {

root html;

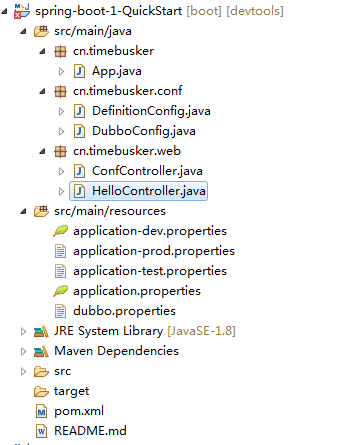
}

}

}

## 3.2 SpringBoot

（使用Spring Boot，无需再做tomcat配置）



## SL4J日志配置

集成logback与SL4J一起使用，在磁盘打印日志，然后收集。

<?xml version="1.0" encoding="UTF-8"?>  
<!--debug="true" : 打印logback内部状态（默认当logback运行出错时才会打印内部状态 ），配置该属性后打印条件如下（同时满足）：   
 1、找到配置文件 2、配置文件是一个格式正确的xml文件 也可编程实现打印内部状态，例如： LoggerContext lc = (LoggerContext)   
 LoggerFactory.getILoggerFactory(); StatusPrinter.print(lc); -->  
<!-- scan="true" ： 自动扫描该配置文件，若有修改则重新加载该配置文件 -->  
<!-- scanPeriod="30 seconds" : 配置自动扫面时间间隔（单位可以是：milliseconds, seconds, minutes   
 or hours，默认为：milliseconds）， 默认为1分钟，scan="true"时该配置才会生效 -->  
<configuration debug="false" scan="true" scanPeriod="30 seconds" packagingData="true">  
 <!-- 设置 logger context 名称,一旦设置不可改变，默认为default -->  
 <contextName>myAppName</contextName>  
  
 <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">  
 <!-- encoders are by default assigned the type ch.qos.logback.classic.encoder.PatternLayoutEncoder -->  
 <encoder>  
 <pattern>%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n</pattern>  
 </encoder>  
 </appender>  
  
 <appender name="FILE" class="ch.qos.logback.core.rolling.RollingFileAppender">  
 <!-- 当前活动日志文件名 -->  
 <file>./my\_log.log</file>  
 <!-- 文件滚动策略根据%d{patter}中的“patter”而定，此处为每天产生一个文件 -->  
 <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">  
 <!-- 归档文件名“.zip或.gz结尾”,表示归档文件自动压缩 -->  
 <FileNamePattern>./my\_log%d{yyyyMMdd}.log.zip</FileNamePattern>  
 <maxHistory>30</maxHistory>  
 </rollingPolicy>  
  
 <!--rollingPolicy class="ch.qos.logback.core.rolling.FixedWindowRollingPolicy">  
 <fileNamePattern>renhai%i.log</fileNamePattern>  
 <minIndex>1</minIndex>  
 <maxIndex>10</maxIndex>  
 </rollingPolicy>  
   
 <triggeringPolicy class="ch.qos.logback.core.rolling.SizeBasedTriggeringPolicy">  
 <maxFileSize>20MB</maxFileSize>  
 </triggeringPolicy-->  
  
 <!-- <rollingPolicy class="ch.qos.logback.core.rolling.SizeAndTimeBasedRollingPolicy">  
 rollover daily  
 <fileNamePattern>mylog-%d{yyyy-MM-dd}.%i.log</fileNamePattern>  
 each file should be at most 30MB, keep 60 days worth of history, but at most 20GB  
 <maxFileSize>30MB</maxFileSize>   
 <maxHistory>60</maxHistory>  
 <totalSizeCap>20GB</totalSizeCap>  
 </rollingPolicy> -->  
  
  
 <encoder>  
 <pattern>%d{HH:mm:ss.SSS}[%-5level][%thread]%logger{36} - %msg%n</pattern>  
 <!-- <pattern>%d{HH:mm:ss.SSS}[%-5level][%thread] - %msg%n</pattern> -->  
 </encoder>  
 </appender>  
  
 <!-- 日志级别若没显示定义，则继承最近的父logger（该logger需显示定义level,直到rootLogger）的日志级别-->  
 <!-- logger的appender默认具有累加性（默认日志输出到当前logger的appender和所有祖先logger的appender中），可通过配置 “additivity”属性修改默认行为-->  
 <logger name="com.yinz" level="debug" additivity="false" >  
 <appender-ref ref="FILE"/>  
 </logger>  
  
 <!-- 至多只能配置一个root -->  
 <root level="debug">  
 <appender-ref ref="STDOUT" />  
 <appender-ref ref="FILE" />  
 </root>  
</configuration>

## 3.4 Flume

### 3.3.1 多层 Flume 部署

数据采集层 Flume 配置

代码清单 3-7 数据采集层Flume配置

a1.sources = r1

a1.channels = c1

a1.sinkgroups = g1

a1.sinks = k1 k2

a1.sources.r1.type = com.atguigu.flume.source.TaildirSource

a1.sources.r1.channels = c1

a1.sources.r1.positionFile = /opt/modules/flume/checkpoint/behavior/taildir\_position.js on

a1.sources.r1.filegroups = f1

a1.sources.r1.filegroups.f1 = /opt/modules/apache-tomcat-7.0.72-1/logs/LogsCollect/atgu igu.log

a1.sources.r1.fileHeader = true

a1.channels.c1.type = file

a1.channels.c1.checkpointDir = /opt/modules/flume/checkpoint/behavior

a1.channels.c1.dataDirs = /opt/modules/flume/data/behavior/

a1.channels.c1.maxFileSize = 104857600

a1.channels.c1.capacity = 90000000

a1.channels.c1.keep-alive = 60

a1.sinkgroups.g1.sinks = k1 k2

a1.sinkgroups.g1.processor.type = load\_balance

a1.sinkgroups.g1.processor.backoff = true

a1.sinkgroups.g1.processor.selector = round\_robin

a1.sinkgroups.g1.processor.selector.maxTimeOut=10000

a1.sinks.k1.type = avro

a1.sinks.k1.channel = c1

a1.sinks.k1.batchSize = 1

a1.sinks.k1.hostname = hadoop-senior02.itguigu.com

a1.sinks.k1.port = 1234

a1.sinks.k2.type = avro

a1.sinks.k2.channel = c1

a1.sinks.k2.batchSize = 1

a1.sinks.k2.hostname = hadoop-senior03.itguigu.com

a1.sinks.k2.port = 1234

1. 聚合层 Flume 配置

代码清单 3-8 聚合层Flume配置

a1.sources = r1

a1.channels = c1

a1.sinks = k1

a1.sources.r1.type = avro

a1.sources.r1.channels = c1

a1.sources.r1.bind = 0.0.0.0

a1.sources.r1.port = 1234

a1.channels.c1.type = file

a1.channels.c1.checkpointDir = /opt/modules/flume/checkpoint/behavior\_collect

a1.channels.c1.dataDirs = /opt/modules/flume/data/behavior\_collect

a1.channels.c1.maxFileSize = 104857600

a1.channels.c1.capacity = 90000000

a1.channels.c1.keep-alive = 60

a1.sinks.k1.type = org.apache.flume.sink.kafka.KafkaSink

a1.sinks.k1.topic = analysis-test

a1.sinks.k1.brokerList = hadoop-senior01.itguigu.com:9092,hadoop-senior02.itguigu.com:9092,hadoop-senior03.itgu igu.com:9092

a1.sinks.k1.requiredAcks = 1

a1.sinks.k1.kafka.producer.type = sync

a1.sinks.k1.batchSize = 1

a1.sinks.k1.channel = c1

### 3.3.2 单层 Flume 部署

代码清单 3-9 单层Flume配置

a1.sources = r1

a1.channels = c1

a1.sinks = k1

a1.sources.r1.type = com.atguigu.flume.source.TaildirSource

a1.sources.r1.channels = c1

a1.sources.r1.positionFile=/opt/modules/flume/checkpoint/behavior/taildir\_position.json

a1.sources.r1.filegroups = f1

a1.sources.r1.filegroups.f1=/opt/modules/apache-tomcat-7.0.72-1/logs/LogsCollect/atgu igu.log

a1.sources.r1.fileHeader = true

a1.channels.c1.type = file

a1.channels.c1.checkpointDir = /opt/modules/flume/checkpoint/behavior\_collect

a1.channels.c1.dataDirs = /opt/modules/flume/data/behavior\_collect

a1.channels.c1.maxFileSize = 104857600

a1.channels.c1.capacity = 90000000

a1.channels.c1.keep-alive = 60

a1.sinks.k1.type = org.apache.flume.sink.kafka.KafkaSink

a1.sinks.k1.topic = analysis-test

a1.sinks.k1.brokerList = hadoop-senior01.itguigu.com:9092,hadoop-senior02.itguigu.com:9092,hadoop-senior03.itgu igu.com:9092

a1.sinks.k1.requiredAcks = 1

a1.sinks.k1.kafka.producer.type = sync

a1.sinks.k1.batchSize = 1

a1.sinks.k1.channel = c1

## 3.5 Kafka

### 3.5.1 创建 Kafka 主题

kafka-topics.sh --zookeeper hadoop102:2181 --create --replication-factor 3 --partitions 1 --topic topic\_app\_startup

## 3.6 Hive

### 3.6.1 Hive 创建数据库及分区表

1. 查看数据库

hive (default)> show databases;

如果 applogs\_db 存在则删除数据库：

hive (default)> drop database applogs\_db;

创建数据库

hive (default)> create database applogsdb;

使用 applogs\_db 数据库：

hive (default)> use applogsdb;

创建分区表

--startup

CREATE external TABLE ext\_startup\_logs(userId string,appPlatform string,appId

string,startTimeInMs bigint,activeTimeInMs bigint,appVersion string,city string)PARTITIONED BY (ym string, day string,hm string) ROW FORMAT SERDE 'org.openx.data.jsonserde.JsonSerDe' STORED AS TEXTFILE;

1. 查看数据库中的分区表

hive (applogsdb)> show tables;

5. 退出 Hive

hive (applogsdb)> quit;

### 3.6.2 Hive 执行脚本

代码清单 3-10 Hive执行脚本

#!/bin/bash

1. 获取三分钟之前的时间

systime=`date -d "-3 minute" +%Y%m-%d-%H%M`

1. 获取年月

ym=`echo ${systime} | awk -F '-' '{print $1}'`

1. 获取日

day=`echo ${systime} | awk -F '-' '{print $2}'`

1. 获取小时分钟

hm=`echo ${systime} | awk -F '-' '{print $3}'`

1. 执行 hive 命令

hive -e "load data inpath '/user/atguigu/test/${ym}/${day}/${hm}' into table loganalysisdb.ext\_startup\_logs partition(ym='${ym}',day='${day}',hm='${hm}')"

## 3.7 Spark SQL

Spark SQL 的命令基本同hive，spark执行脚本的命令

spark-sql -e "$sql" --driver-java-options "-Dlog4j.configuration=file:///mnt/log4j.properties"

## 3.8 Logstah

（未完）

# 第4章 代码和脚本解析

## 4.1 离线数据处理系统

### 4.1.1 Flume读取Kafka数据

离线数据处理系统中的 Kafka 高级消费者程序将消息从 Kafka 集群中消费出来，然后写入指定的 HDFS 文件中。采用lzo压缩文件

代码清单 4-1 Flume 读取kafka的数据

agent.sources=r1

agent.sinks=k1

agent.channels=c1

agent.sources.r1.type= org.apache.flume.source.kafka.KafkaSource

agent.sources.r1.kafka.bootstrap.servers=localhost:9092

agent.sources.r1.kafka.topics=test

#agent.sources.r1.serializer.class=kafka.serializer.StringEncoder

agent.sources.r1.kafka.consumer.group.id = flumetest

agent.sources.r1.flumeBatchSize=1000

agent.sources.r1.useFlumeEventFormat=false

agent.sources.r1.restart=true

agent.sources.r1.batchSize=1000

agent.sources.r1.batchTimeout=3000

agent.sources.r1.channels=c1

agent.channels.c1.type=memory

agent.channels.c1.capacity=102400

agent.channels.c1.transactionCapacity=1000

agent.channels.c1.byteCapacity=134217728

agent.channels.c1.byteCapacityBufferPercentage=80

agent.sinks.k1.channel=c1

agent.sinks.k1.type=hdfs

agent.sinks.k1.hdfs.path=hdfs:///tmp/flume/%Y%m%d

agent.sinks.k1.hdfs.writeFormat=Text

agent.sinks.k1.hdfs.rollSize=0

agent.sinks.k1.hdfs.rollCount=0

agent.sinks.k1.hdfs.rollInterval=120

agent.sinks.k1.hdfs.threadsPoolSize=30

agent.sinks.k1.hdfs.fileType = CompressedStream

agent.sinks.k1.hdfs.fileSuffix=.lzo

agent.sinks.k1.hdfs.codeC = lzop

### 4.1.2 Hive函数

编写udf函数

1. 聚合函数，找出非空的值

*/\*\*  
 \* 判断不是 null和空串这两种情况  
 \*/*public class NotNullValueUDAF extends AbstractGenericUDAFResolver {  
 static final Log *LOG* = LogFactory.*getLog*(NotNullValueUDAF.class.getName());  
  
 public GenericUDAFEvaluator getEvaluator(GenericUDAFParameterInfo info) throws SemanticException {  
 return new GenericUDAFHistogramNumericEvaluator();  
 }  
  
 public static class GenericUDAFHistogramNumericEvaluator extends GenericUDAFEvaluator {  
  
 private PrimitiveObjectInspector inputOI;  
  
 //这个方法返回了UDAF的返回类型，这里确定了sum自定义函数的返回类型是Long类型  
 @Override  
 public ObjectInspector init(Mode m, ObjectInspector[] parameters) throws HiveException {  
 assert (parameters.length == 1);  
 super.init(m, parameters);  
 inputOI = (PrimitiveObjectInspector) parameters[0];  
 return PrimitiveObjectInspectorFactory.*javaStringObjectInspector*;  
 }  
 @Override  
 public AbstractAggregationBuffer getNewAggregationBuffer(){  
 NotNullAgg buffer = new NotNullAgg();  
 reset(buffer);  
 return buffer;  
 }  
  
 @Override  
 public void reset(AggregationBuffer agg) {  
 NotNullAgg notNullAgg=(NotNullAgg)agg;  
 notNullAgg.val="";  
 }  
  
 @Override  
 public void iterate(AggregationBuffer agg, Object[] parameters) {  
 assert (parameters.length == 1);  
 merge(agg,parameters[0]);  
 }  
  
 @Override  
 public Object terminatePartial(AggregationBuffer agg) {  
 return terminate(agg);  
 }  
  
 @Override  
 public void merge(AggregationBuffer agg, Object partial) {  
 if(partial!=null){  
 NotNullAgg notNullAgg=(NotNullAgg)agg;  
 String input= PrimitiveObjectInspectorUtils.*getString*(partial, inputOI).trim();  
 if(StringUtils.*isNotBlank*(input)){  
// 还要判断是否存在 null nul 这两种情况  
 if(!input.equalsIgnoreCase("null") && ! input.equalsIgnoreCase("none")) {  
 notNullAgg.val = input;  
 }  
 }  
 }  
 }  
  
 @Override  
 public Object terminate(AggregationBuffer agg) {  
 NotNullAgg myagg = (NotNullAgg) agg;  
 return new Text(myagg.val);  
 }  
  
 }  
  
 @GenericUDAFEvaluator.AggregationType(estimable = true)  
 static class NotNullAgg extends GenericUDAFEvaluator.AbstractAggregationBuffer {  
 String val="";  
 }

1. UDTF 一列转多列

import java.util.ArrayList;  
  
import org.apache.commons.lang3.StringUtils;  
import org.apache.hadoop.hive.ql.exec.UDFArgumentException;  
import org.apache.hadoop.hive.ql.metadata.HiveException;  
import org.apache.hadoop.hive.ql.udf.generic.GenericUDTF;  
import org.apache.hadoop.hive.serde2.objectinspector.ObjectInspector;  
import org.apache.hadoop.hive.serde2.objectinspector.ObjectInspectorFactory;  
import org.apache.hadoop.hive.serde2.objectinspector.StructObjectInspector;  
import org.apache.hadoop.hive.serde2.objectinspector.primitive.PrimitiveObjectInspectorFactory;  
import org.json.JSONArray;  
import org.json.JSONException;  
  
*/\*\*  
 \* 用于将json串中的json数组展平，将eventname字段提取出来，并且保存成event\_name 和event\_json  
 \*  
 \** ***@author*** *tomluo  
 \*  
 \*/*public class EventnameListFlatter extends GenericUDTF {  
  
 public static final String *EVENT\_NAME\_FIELD* = "eventname";  
  
 @Override  
 public StructObjectInspector initialize(StructObjectInspector argOIs) throws UDFArgumentException {  
  
 ArrayList<String> fieldNames = new ArrayList<String>();  
 ArrayList<ObjectInspector> fieldOIs = new ArrayList<ObjectInspector>();  
 fieldNames.add("event\_name");  
 fieldOIs.add(PrimitiveObjectInspectorFactory.*javaStringObjectInspector*);  
 fieldNames.add("event\_json");  
 fieldOIs.add(PrimitiveObjectInspectorFactory.*javaStringObjectInspector*);  
 return ObjectInspectorFactory.*getStandardStructObjectInspector*(fieldNames, fieldOIs);  
 }  
  
 @Override  
 public void process(Object[] objects) throws HiveException {  
 if (objects == null || objects[0] == null) {  
 forward(new Object[] { null, null });  
 return;  
 }  
 String input = objects[0].toString();  
 JSONArray ja = null;  
 if (StringUtils.*isNotBlank*(input)) {  
 try {  
 ja = new JSONArray(input);  
 } catch (JSONException e) {  
 e.printStackTrace();  
 }  
 if (ja == null)  
 return;  
 for (int i = 0; i < ja.length(); i++) {  
 String[] result = new String[2];  
 try {  
 result[0] = ja.getJSONObject(i).getString(*EVENT\_NAME\_FIELD*);  
 result[1] = ja.getString(i);  
 } catch (JSONException e) {  
 continue;  
 }  
 forward(result);  
 }  
 } else {  
 forward(new Object[] { "", "" });  
 }  
 }  
  
 @Override  
 public void close() throws HiveException {  
  
 }  
  
}

### 4.1.3 Hive流程语句

1.创建外部表，连接原始

-- 创建外部表，连接原始日志，对应数仓的原始数据层

CREATE EXTERNAL TABLE `stage\_originlog\_lzo\_dt`(`line` string)  
PARTITIONED BY (`dt` string)  
ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe'  
WITH SERDEPROPERTIES (  
 'serialization.format' = '1'  
)  
STORED AS  
 INPUTFORMAT 'com.hadoop.mapred.DeprecatedLzoTextInputFormat'  
 OUTPUTFORMAT 'org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat'  
LOCATION 'hdfs:/tmp/flume/';、

– 添加分区，让数据可以查看

alter table stage\_originlog\_lzo\_dt add partition (dt='2018-09-01') location 'hdfs:/tmp/flume/20180901/';

1. 创建细节表，一行转多行，同时提取数据

use test;  
add jar /home/hadoop/teach.jar;  
create temporary function base\_analizer as 'com.test.BaseFieldUDF';  
create temporary function flat\_analizer as 'com.test.EventnameUDTF';  
set hive.exec.dynamic.partition.mode=nonstrict;  
  
insert overwrite table ods\_basedata\_dt  
PARTITION (dt)  
select  
 user\_id ,   
 area ,   
 app\_version ,   
 app\_time ,   
 event\_name ,   
 event\_json ,  
 server\_time ,   
 sdk\_log.dt   
 from  
(  
select  
split(base\_analizer(line,'userId,area,appVersion,appTime'),'\t')[0] as user\_id ,  
split(base\_analizer(line,'userId,area,appVersion,appTime'),'\t')[1] as area ,  
split(base\_analizer(line,'userId,area,appVersion,appTime'),'\t')[2] as app\_version ,  
split(base\_analizer(line,'userId,area,appVersion,appTime'),'\t')[3] as app\_time ,  
split(base\_analizer(line,'userId,area,appVersion,appTime'),'\t')[4] as server\_time ,  
split(base\_analizer(line,'userId,area,appVersion,appTime'),'\t')[5] as ops ,  
dt  
from stage\_originlog\_lzo\_dt where dt='2018-09-01'   
) sdk\_log lateral view flat\_analizer(ops) tmp\_k as event\_name, event\_json;

创建用户表

CREATE EXTERNAL TABLE `dw\_history\_users`(`uid` string COMMENT 'uid', `area` string COMMENT 'area',   
`first\_dat` string COMMENT 'first appear time', `first\_version` string COMMENT 'first appear version,default 0.0.0',   
 `current\_dat` string COMMENT 'latest time', `current\_version` string COMMENT 'latest version')  
PARTITIONED BY (`dt` string);

### 4.1.4 Mongo流程语句

1.创建新闻表

db.createCollection('user\_news')

2.结构,\_id为（newsid+user\_id）

{"\_id":news\_id+user\_id, "create\_time":记录的时间，用long类型}

3.创建时间索引

db.user\_news.ensureIndex('create\_time')

## 4.2实时数据处理系统

### 4.2.1 SparkStreaming程序

流式计算做的事情是

1. 计算出新闻的热度。
2. 将新闻排序

import com.mongodb.spark.MongoSpark;  
import org.apache.kafka.clients.consumer.ConsumerRecord;  
import org.apache.spark.SparkConf;  
import org.apache.spark.api.java.JavaRDD;  
import org.apache.spark.api.java.function.Function0;  
import org.apache.spark.streaming.Durations;  
import org.apache.spark.streaming.api.java.JavaInputDStream;  
import org.apache.spark.streaming.api.java.JavaStreamingContext;  
import org.apache.spark.streaming.kafka010.\*;  
import org.bson.Document;  
  
import java.util.\*;  
  
  
public class ReadNewsStream {  
 public static void main(String[] args) throws Exception {  
 final Properties serverProps = PropertiesUtils.*getProperties*("config.properties");  
//获取 checkpoint 的 hdfs 路径  
 String checkpointPath = serverProps.getProperty("streaming.checkpoint.path");  
// 如果 checkpointPath hdfs 目录下的有文件，则反序列化文件生产 context, 否则使用函数 createContext 返回的 context 对象  
 JavaStreamingContext javaStreamingContext = JavaStreamingContext.*getOrCreate*(checkpointPath, *createContext*(serverProps));  
 javaStreamingContext.start();  
 javaStreamingContext.awaitTermination();  
 }  
  
 */\*\*  
 \* 根据配置文件以及业务逻辑创建 JavaStreamingContext  
 \*  
 \** ***@param*** *serverProps  
 \** ***@return*** *\*/* public static Function0<JavaStreamingContext> createContext(final Properties serverProps) {  
 Function0<JavaStreamingContext> createContextFunc = new Function0<JavaStreamingContext>() {  
 public JavaStreamingContext call() throws Exception {  
//获取配置中的 topic  
 String topicStr = serverProps.getProperty("kafka.topic");  
 Collection<String> topics = Arrays.*asList*(topicStr.split(","));  
 //获取配置中的 groupId  
 final String groupId = serverProps.getProperty("kafka.groupId");  
//获取批次的时间间隔，比如 5s  
 final Long streamingInterval = Long.*parseLong*(serverProps.getProperty("streaming.interval"));  
//获取 checkpoint 的 hdfs 路径  
 final String checkpointPath = serverProps.getProperty("streaming.checkpoint.path");  
//获取 kafka broker 列表  
 final String kafkaBrokerList = serverProps.getProperty("kafka.broker.list");  
//组合 kafka 参数  
 final Map<String, Object> kafkaParams = new HashMap();  
 kafkaParams.put("metadata.broker.list", kafkaBrokerList);  
 kafkaParams.put("group.id", groupId);  
  
// 创建 SparkConf 对象  
 SparkConf sparkConf = new SparkConf().setMaster("local[\*]").setAppName("spark-news");  
  
*/\*\*  
 优雅停止 Spark. 暴力停掉 sparkstreaming 是有可能出现问题的，比如你的数据源是 kafka，  
 已经加载了一批数据到 sparkstreaming 中正在处理，如果中途停掉，  
 这个批次的数据很有可能没有处理完，就被强制 stop 了，  
 下次启动时候会重复消费或者部分数据丢失。  
 \*/* sparkConf.set("spark.streaming.stopGracefullyOnShutdown", "true");  
  
 */\*\*在 Spark 的架构中，在网络中传递的或者缓存在内存、硬盘中的对象需要进行序列化操作，序列化的作用主要是利用时间换空间\*/* sparkConf.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer");  
  
  
 */\*\*增加 MyRegistrator 类，注册需要用 Kryo 序列化的类,Kryo 的序列化会比java的要更省空间\*/*// sparkConf.set("spark.kryo.registrator", "com.tom.MyKryoRegisrator");  
  
  
 */\*\* 每秒钟对于每个 partition 读取多少条数据如果不进行设置，Spark Streaming 会一开始就读取 partition 中的所有数据到内存，给内存造成巨大压力  
 设置此参数后可以很好地控制 Spark Streaming 读取的数据量，也可以说控制了读取的进度 \*/* sparkConf.set("spark.streaming.kafka.maxRatePerPartition", "100");  
  
 /\* 创建 javaStreamingContext，设置 每隔5s 执行一次\*/  
 JavaStreamingContext javaStreamingContext = new JavaStreamingContext(sparkConf, Durations.*seconds*(streamingInterval));  
 javaStreamingContext.checkpoint(checkpointPath);  
//创建 kafka DStream  
 final JavaInputDStream<ConsumerRecord<String, String>> kafkaMessage = KafkaUtils.*createDirectStream*(javaStreamingContext, LocationStrategies.*PreferConsistent*(),  
 ConsumerStrategies.<String, String>*Subscribe*(topics, kafkaParams));  
  
//需要把每个批次的 offset 保存  
 kafkaMessage.foreachRDD(rdd -> {  
 /\*表示具有[[OffsetRange]]集合的任何对象，这可以用来访问由直 Direct Kafka DStream 生成的 RDD 中的偏移量范围\*/  
 OffsetRange[] offsetRanges = ((HasOffsetRanges) rdd.rdd()).offsetRanges();  
 /\*逻辑处理在这里\*/  
 *executeData*(rdd);  
 /\*kafka offset 写入 zk\*/  
 ((CanCommitOffsets) kafkaMessage.inputDStream()).commitAsync(offsetRanges);  
 });  
//将 kafka 中的消息转换成对象并过滤不合法的消息  
 return javaStreamingContext;  
 }  
 };  
 return createContextFunc;  
 }  
  
 static private void executeData(JavaRDD<ConsumerRecord<String, String>> rdd) {  
//1.计算威尔逊热度  
 JavaRDD<String> result = rdd.filter(stringStringConsumerRecord -> {  
 /\*过滤出来只有点击的日志\*/  
 String content = stringStringConsumerRecord.value();  
 return content.contains("action");  
 }).map(stringStringConsumerRecord -> {  
 /\*组装成新闻的点击格式\*/  
 Document document = Document.parse();  
 return stringStringConsumerRecord.value();  
 });  
// 2.写入mongoDB  
// MongoSpark.save(result);  
 }  
}

# 第5章 项目调试与运行

## 5.1 Nginx负载均衡

### 5.1.1 Nginx启动

sudo /usr/local/webserver/nginx/sbin/nginx -c /opt/modules/nginx-1.12.2/conf/nginx.conf

### 5.1.2 SpringBoot启动

mvn install

cd target

java -jar xxxx.jar

## 5.2 公共日志采集系统

### 5.2.1 Zookeeper集群启动

ZkServer.sh start

### 5.2.2 数据采集层Flume启动

flume-ng agent --classpath /opt/modules/flume/lib/flume-taildirsource.jar --conf /opt/modules/flume/conf/ -f /opt/modules/flume/conf/flume-analysis.conf -n a1

### 5.2.3 Kafka 集群启动

kafka-server-start.sh –daemon config/server.properties

## 5.3 离线系统

### 5.3.1 Hadoop集群启动

start-dfs.sh

start-yarn.sh

### 5.3.2 Hive数据仓库启动

hive –-service hiveserver2

hive –-service metastore

hive

### 5.3.3 调度任务启动

service crond start

### 5.3.4 新增用户统计

select count(uid) from dw\_history\_users where dt='2018-09-26' and current\_dat=dt

### 5.3.5 每日新闻展示总量，平均新闻展示，新闻点击，点击率计算

select sum(news\_id) news\_total,sum(display)/sum(news\_id) avg\_display,sum(click)/sum(news\_id) avg\_click, sum(display)/sum(click) avg\_rate  
from (  
select  
 t1.display display,  
 case when t2.click is not null then t2.click else 0 end click,  
 t1.news\_id  
 from (select count(distinct user\_id) display,news\_id from user\_display group by news\_id) t1 left join  
(select count(distinct user\_id ) click,news\_id from user\_click group by news\_id) t2 on  
t1.news\_id=t2.news\_id  
) t3;

### 5.3.6 前台，后台活跃用户数统计

select t1.forground\_total,t2.background\_total,t1.dt from  
(select count(distinct uid) forground\_total,dt from dw\_foreground where dt='2018-09-25' group by dt) t1  
join (select count(distinct uid) background\_total,dt from dw\_background where dt='2018-09-25' group by dt) t2  
on t1.dt=t2.dt ;

### 5.3.7 留存用户统计

### 5.3.8 沉默用户统计

### 5.3.9 每日历史用户分析

### 5.4.0 每日历史新闻分析

## 5.4 实时系统

### 5.4.1 MongoDB启动

mongodb/bin/mongod --dbpath mongodb/0706 --fork

### 5.4.2 SparkStreaming 程序启动

spark-submit \

--class com.tom.spark.SparkNewsCaculator \

--master yarn \

--deploy-mode cluster \

--queue migsz \

--driver-memory 5G \

--executor-memory 10G \

--num-executors 10 \

--executor-cores 4 \

--conf spark.shuffle.io.maxRetries=6

Spark-news.jar

### 5.4.3 实时计算新闻热度

威尔逊函数+阈值，然后排序质量新闻和热度新闻。调用算法函数。

（未完）

# 第6章 项目总结

1. 用户行为数据通过http接口上报到web服务器，web服务型将上报的用户行为数据写入本地磁盘，由此实现了业务与数据处理的解耦，之后再由Flume实时监控文件并进行采集。
2. Flume数据采集构架采用双层Flume拓扑架构，第一层实现数据的采集，第二层实现数据的聚合，并通过Flume拓扑结构中的Sinkgroup实现了Load Balance，充分利用了资源。
3. 第一层Flumeagent在每台web服务器上部署一个，当挂掉重启后不能丢失数据，所以我们用了TaildirSource数据源，此数据源会记录每个文件采集到的位置，重启后会从记录的位置采集，但是此数据源有缺陷(当文件重命名后会重复采集)，故我们针对此缺陷进行了源码修改。
4. 第二层聚合Flumeagent需要部署2个及以上，我们的实战项目中有两个聚合agent,其中一个挂掉后，系统可以继续运行，不丢数据，不影响结果。
5. 第二层聚合Flumeagent将数据写入kafka, 当由于某种原因比如网络问题或者kafka停服不能写入时，数据会在第二层agent中的channel中累积，不影响第一层agent的采集，当kafka能够写入时，kafkasink会继续消费第二层angent的channel中的数据。
6. 当第二层聚合Flumeagent全部挂掉时，数据会在第一层的采集agent中的channel中累积，不影响数据采集。
7. Spark Streaming需要7x24小时运行，一旦Driver挂掉后能够自动重启,我们让其在YARN中以Cluster模式运行，此时Driver运行在ApplicationMaster中，当ApplicationMaster挂掉后会重新启动。
8. 为了Driver重启后需要能恢复到之前的状态，所以我们要设置Checkpoint目录，以持久化Kafka offset、未完成的job等到HDFS。
9. 当Spark Streaming的代码修改后启动时，反序列化Checkpoint目录中的数据失败，所以Kafkaoffset会丢失，此时不知道从哪里消费Kafka的数据，所以我们要将Kafka offset保存到ZooKeeper中一份，当Spark Streaming优雅停止后，删除Checkpoint目录然后从ZooKeeper中读取Kafka offset再启动SparkSteaming。